Contents

1 Inngangur **3**

2 Sjálfvirk niðurþrýstingsvél **9**

3 Sjálfvirkar ýtingarvélar og Turing vélar **19**

4 Kynning á niðurdrepandi uppteknum bever **31**

5 Turing heilleika Pushdown sjálfvirkum aðgerðum og útreikningarmöguleikum pudau **43**

6 Niðurstöðu **67**

7 Endalok **69**

Pudau - Puhoy

Kumenonhi (Supersubstantial 3301)

2 July 2025

Chunck 1

Inngangur

pushdown automata: Formal definition and computational limits 1. Overview of Automata Theory and Pushdown Automata 1.1. Summary of Computational Models The domain of theoretical computer science utilizes abstract constructs called automata to represent computation and language identification. These models offer a fundamental insight into what is computable, the efficiency of the processes, and the resources required. The exploration of automata theory is essential for understanding the fundamental strengths and weaknesses of different computational models, providing the foundation for fields like compiler development, algorithm evaluation, and the essence of computation itself. Every computational model has a unique degree of computational power that determines the types of problems it can address. The Chomsky Hierarchy: A System for Classifying Formal Languages and Automata A fundamental aspect of formal language theory is the Chomsky Hierarchy, a classification structure that arranges formal grammars and the languages they produce according to their growing complexity. This hierarchy also creates a direct link between these language categories and the abstract automata that can recognize them. The hierarchy outlines four main categories of grammars and their related languages: Type-3 Grammars produce Regular Lan-

guages. Type-2 Grammars produce Context-Free Languages. Type-1 Grammars produce Context-Sensitive Languages. Type-0 Grammars produce Recursively Enumerable Languages. A key characteristic of this hierarchy is that every class contains the lower ones, indicating a clear enhancement in expressive power and computational ability. Similarly, every language category is linked to a minimal computational framework necessary for its identification: Finite-State Automata (FSA) for Regular Languages, Non-deterministic Pushdown Automata (NPDA) for Context-Free Languages, Linear-Bounded Automata (LBA) for Context-Sensitive Languages, and the most potent, Turing Machines (TM), for Recursively Enumerable Languages. This organized categorization offers a guide for comprehending the computational environment. To exemplify this connection, examine the table below: Grammar Type Language Category Identification Automaton Memory Structure Type-3 (Regular) Regular Languages Finite-State Automaton (FSA) Finite states (no additional memory) Type-2 (Context-Free) Context-Free Languages Non-deterministic Pushdown Automaton (NPDA) Finite states + limitless LIFO stack Type-1 (Context-Sensitive) Context-Sensitive Languages Linear-Bounded Automaton (LBA) Finite states + limited tape (all entries reachable) Type-0 (Recursively Enumerable) Recursively Enumerable Languages Turing Machine (TM) Finite states + limitless random-access tape Ekspor ke Spreadsheet 1.3. The positioning of Pushdown Automata in the Hierarchy Pushdown Automata (PDAs) signifies a significant improvement over Finite Automata adding an extra memory element. This enhancement greatly improves their processing abilities. A PDA is a computational model that enhances the capabilities of Finite Automata including an infinite memory component referred to as a stack. In contrast to Finite Automata, which are restricted to a limited amount of memory represented only their states, PDAs utilize this stack. The stack functions based on a Last-In, First-Out (LIFO) principle, allowing the PDA to monitor and handle an unlimited quantity of data. This abil-

ity makes PDAs especially proficient at identifying Context-Free Languages (CFLs), a category of languages defined nested structures, like those present in programming language syntax (e.g., matched parentheses, function invocations). As a result, PDAs are crucial instruments for syntax analysis in compilers. Preliminary Thoughts and Importance The addition of an unbounded stack memory in Pushdown Automata, although a major improvement compared to Finite Automata, brings about both substantial strength and intrinsic constraints. The stack's particular access method—its LIFO principle—significantly determines these limits. The Chomsky Hierarchy clearly illustrates a direct relationship between the complexity of a computational model's memory and its expressive capability. Finite Automata, which have no external memory apart from their limited set of states, can only recognize the most basic category of languages, referred to as Regular Languages. The addition of a stack in Pushdown Automata, offering a type of limitless memory, instantly raises their ability to identify Context-Free Languages. At the top of this hierarchy, Turing Machines have an infinite tape with random access, providing them with universal computing power. This development highlights a core tenet in automata design: the ability of an automaton to save and retrieve data, and importantly, the way it can access that data, directly influences the category of languages it can identify and, consequently, its total computational power. This causal connection suggests that the architectural decisions in crafting computational models represent a conscious equilibrium between the required computational capability and the limitations and intricacies of the foundational memory system. Moreover, although Turing Machines function as the theoretical standard for universal computation, PDAs still hold considerable practical value. Studies show that PDAs are extremely valuable because "they impose more restrictions than Turing machines" and act as "crucial instruments for syntax analysis in compilers." This observation highlights an important practical aspect of theoretical computer

science: not every computational problem requires the complete, unrestricted capacity of a Turing Machine. For particular, clearly defined tasks, a more restricted model such as a PDA presents clear benefits, comprising easier analysis, more consistent performance traits, and frequently, simpler implementation. The "limitless complexity" and fundamental undecidability problems often linked to Turing Machines can be avoided using a less powerful but adequately capable model. This indicates a dominant strategy where theoretical models are chosen according to the specific needs of a particular issue and the intended operational attributes (like decidability or efficiency), instead of purely focusing on maximizing computational capability. Formal Definition of Pushdown Automata 2.1. The 7-Tuple Definition A Pushdown Automaton (PDA) is a clearly specified mathematical model. A PDA (M) is formally defined as a 7-tuple: M=(Q,Σ,Γ, ,q 0 ,Z 0 ,F). Every element of this tuple serves an essential function. in outlining the actions and functions of the automaton. Components in Detail (Q): Limited Collection of States. This element signifies the finite control unit of the automaton, which can exist in any one of a limited number of states. It serves as the fundamental processing unit that determines the PDA's actions according to the input and the contents of the stack. (Σ) : $InputAlphabet.Thisisalimitedcollectionofsymbolsthatthe PDAcaninterpretfrom$ $StackAlphabet.Thisisalimitedcollectionofsymbolsthatcanbeaddedtoorremovedfr$ $FunctionofTransition.Thisconstitutestheoperationalcenterofthe PDA, outlinin$ $deterministic)PDA, thetransitionfunction(\delta)associatesatriplemadeupofthecurr$ $: Q()P(Q).Inthiscontext, representsthepowerset, signifyingthatforanyinput, sta$ $determinism.WhenthePDAisinstate(q), processesinputsymbol(a)(or(\varepsilon)), and(X)$ $If(\gamma = \varepsilon), thesymbol(X)isremovedfromthestack.If(\gamma = X), thesymbol(X)doesn$ $operationonthetopofthestack).If(\gamma = Y), (Y)ispushedand(X)ispopped.If(\gamma =$ $YZ), (X)isremoved, andthen(YZ)isaddedtothestack, effectivelysubstituting(X)$ $InitialState.Thisisthedistinctstartingstatefromwhichthe PDAinitiatesitscompu$ $StartingStackSymbol.Thissymbolisfirstpositionedatthebottomofthestackwhenti$ $\Gamma).(F) : CollectionofAcceptingStates.Thisisaportionof(Q)thatincludesallstates$

$(q \in Q) : Indicatesthepresentstateofthefinitecontrolunit.(w \in \Sigma^*) :$
$Signifiestheunprocessedpartoftheinputstring.(\alpha \in \Gamma^*) : Denotesthecurrentstackconte$
$).Forexample, ifthePDAisinstate(q), processestheinputsymbol(a), and(X)isatthetopo$
$(p, w, \gamma\beta)).Inthiscase, (a)denotestheconsumedinputsymbol, (X)indicatesthereplacedsta$
$).AcceptanceCriteriaAPushdownAutomatonhastwomain, separatemodesforacceptingc$
$AcceptanceFinalState : Inthismode, aninputstring(w)isacceptedifthePDAreachesasta$
$L(M) = w(q0, w, Z0)(qf, ,)forsomeqfF, Thisstandardisfrequentlyeasiertograspforva$
$Aninputstring(w)isacceptedifthePDA'sstackisentirelyemptyafterithasprocessedthef$
$L(M) = w(q0, w, Z0)(q, ,)forsomeqQ.Thisapproachef fectivelyrepresentslanguagesth$
$deterministicPDAs.ForNon-deterministicPushdownAutomata(NPDAs), thesetwoa$
$freelanguages.Thelackofequivalencebetweentheacceptancefinalstateandtheemptystac$
$deterministicPDAinherentlyhastheabilitytotraverseseveralcomputationalpathsatonce$
$inflexibilityenablesanNPDAto"anticipate"therighttimetoclearitsstackexactlyasitmou$
$deterministically"compel"itsstacktoemptypreciselywhenitarrivesatafinalstate, orthec$
$determinismoffersanessentiallevelofadaptabilitythatallowsforamorecohesiveandcom$
$Context-FreeLanguages3.1.TheconnectionbetweenPushdownAutomataandContext-$
$FreeGrammars(CFGs)isessential, establishingafoundationalequivalencethatispivota$
$FreeLanguages(CFLs).Alanguageissaidtobecontext-freeifandonlyifitcanbeaccepted$
$FreeLanguagesthattheycanacknowledge : LanguagewithEqualCountsof'a'sand'b's :$
$Thelanguage(L = a^nb^n|n0)servesasatraditionalillustrationofaCFL.APDAidentifiest$
$Languagesthatfeaturebalancedformations, likeexpressionswithproperlynestedparenth$
$Thelanguage(L = ww^R \mid w \in \Sigma^*), madeupofstringsthatarepalindromeswithevenlength$
$FreeLanguage.Thislanguagecanbeidentifiedasa Non-deterministicPushdownAutoma$
$deterministically"predicting"themidpointoftheinputstring.Itplacesthecharactersfrom$
$a^nb^{2n} \mid n1) : Thislanguagenecessitatesaparticularproportionofcharacters.Apersonaldi$
$DeterministicPowerThedifferencebetweenDeterministicPushdownAutomata(DPDA$
$deterministicPushdownAutomata(NPDAs)holdssubstantialimportance, bothintheory$
$FreeLanguages, referredtoasDeterministicContext-FreeLanguages(DCFLs).Thisi$
$FreeLanguagesthatcanbeidentifiedanNPDAbutcannotberecognizedanyDPDA.Thisco$
$worldscenarios, likeimprovingconstraineddecodinginLargeLanguageModels(LLMs),$
$deterministictransitions, likebacktracking, speculativetasks, andmanagingintricateper$
$deterministically"predict"themidpointoftheinputstring.isanessentialpartofitsidentif$

$deterministicdecision.Torecognizethistypeoflanguage, aDPDAwouldneedadeter$
$like"decision, isanessentialrequirementforrecognition.Thisdistinctiondeeplyhig$
$determinismprovidesaconsiderablecomputationalbenefittoPDAs, allowingthemt$
$FreeLanguages.Theincorporationofanunlimitedstackgrantsthe PDAmemoryfu$

The reverse is incorrect: a standard Pushdown Automaton is unable to simulate any arbitrary Turing Machine. This imbalance is the fundamental reason that PDAs lack Turing completeness. The computational dominance of Turing Machines is so evident that even a severely limited one-state Turing machine has proven to be more powerful than a typical Pushdown Automaton, able to recognize languages that are beyond context-free. This illustrates a specific instance of the TM's enhanced computational capability, even in simple setups, compared to PDAs. 4.2. Intrinsic Constraints of a Singular Stack (LIFO Principle) The Last-In, First-Out (LIFO) principle inherent in a PDA's single stack allows for unbounded memory but introduces certain limitations that restrict PDAs from recognizing languages outside the context-free category. These constraints arise from the limited access to the stack's elements: solely the uppermost symbol is immediately accessible. Restricted Independent Comparisons: A PDA, utilizing its singular stack, can proficiently carry out comparisons or keep counts solely between two associated segments of a string. For example, in languages such as ($\mathrm{L} = \mathrm{a}^n b^n$), $thePDAcaninsert'a'sandthenremovethemfor'b's, there$
$a^n b^n c^n | n0).APDAcaneffectivelypair(a^n)with(b^n)pushing'a'sandpoppingfor'b's.T$
$afterthe'a'sareremovedtocorrespondwiththe'b's, theinitialnumberof'a'sisnolonge$
$Comparison : Languagesthatrequire"cross-comparison,"whereelementsrequir$
$stackPDA.Forinstance, thelanguage(L = a^m b^n c^m d^n)isnotrecognizable.Toassess($
$of-StackData : TheLIFOcharacteristicofthestackindicatesthattheinitialsymb$
$WW \mid W \in \Sigma)(suchas"abababab")isnotcontext-free.Whenthefirst(W)isplacedo$

Chunck 2

Sjálfvirk niðurþrýstingsvél

Formlega skilgreiningin á niðurþrýstingsvélum Niðurþrýstingsvélar (PDAs) eru flokkur af huglægum vélar sem auka hæfni endanlegra véla (FAs) með því að bæta við stafla. This extra memory structure, a Last-In, First-Out (LIFO) data store, gives PDAs the essential capacity to retain an unlimited quantity of details regarding the input string's past. This improved memory enables PDAs to identify context-free languages, which constitute a wider and more intricate category of languages than the regular languages recognized FAs. The official definition of a PDA is not just an intellectual pursuit; it is essential for grasping its computational capabilities, its functioning, and essentially, how it establishes the foundation for the theory of parsing and compiler development. In fact, PDAs function as the exact theoretical framework for parsers that assess the syntax of programming languages, which are fundamentally context-free and necessitate memory beyond what a finite automaton can supply to manage nested structures such as parentheses, block scopes, and function invocations. A Pushdown Automaton is rigorously specified as a 7-tuple: M=(Q,Σ,Γ, ,q 0 ,Z 0

,F). We will analyze each part of this tuple in thorough detail, examining their functions and significance in the PDA's functioning: Q: A finite, non-empty collection of states. This element corresponds directly to the collection of states in a Finite Automaton, symbolizing the internal arrangement of the automaton's control unit. In a PDA, the "significance" of a state is frequently more complex because of its relationship with the stack. Every state signifies a particular arrangement or phase of the automaton's computation, frequently indicating which portion of the input has been handled, which grammatical rule is presently in use, or which structural component of the language is being acknowledged at that moment. For example, in a PDA created to identify balanced parentheses, states could indicate "waiting for an opening parenthesis," "within a parenthesis segment," or "completed handling a balanced portion." The shifts between these states are controlled not only the input but also the symbol at the top of the stack. Q=q1, q2, ..., qn Σ: A non-empty, finite collection of symbols from the input alphabet. These are the basic elements that the PDA processes from its input tape, one at a time, from left to right. This collection specifies the characters or tokens that can create the strings of the language acknowledged the PDA. For a PDA that recognizes a specific subset of natural language, Σ may encompass singular letters, punctuation marks, or even complete words. Within the framework

In programming languages, Σ usually denotes the collection of lexical tokens (for instance, IF, ELSE, WHILE, identifiers, and operators such as +, -, *, parentheses (,)). The PDA processes these symbols in order, determining actions based on the present input symbol, its internal state, and the top of the stack. Σ=a1, a2, ..., am Γ: A finite, non-empty collection of symbols for the stack alphabet. These are the symbols that can be added to or removed from the stack. It is important to recognize that Σ and Γ are different sets, even if they might have some symbols in common. For instance, the input alphabet Σ could include '(' and ')', whereas the

stack alphabet Γ may consist of these same symbols, along with additional markers such as X, Y, or a unique bottom-of-stack symbol like $.Thestacksymbolsresidewithin the PDA's memory and serve to monitor structural details o$ $to-one relationship. = 1, 2, , k : , PDA, .It determines how the PDA transitions from one c$ $deterministic nature. This non-determinism is an essential characteristic that provides P$ $Q()P(Q) Let's analyze the elements of$

To fully understand its functionality, consider the transition function's domain and codomain: The domain Q×(Σ)×Γ outlines the exact conditions for a transition to take place: The current state q Q: This indicates the state in which the PDA currently resides. The input symbol a Σ, or : This denotes the symbol that is being examined from the input tape. The incorporation of (the empty string) is essential. An -transition permits the PDA to transition without using an input symbol from the tape. These "null transitions" are potent, allowing the PDA to adjust its stack or switch states solely based on its internal state and stack contents, without moving the input pointer forward. This is especially beneficial for managing cases where a grammatical rule suggests a stack operation without a corresponding input symbol, or for modeling non-deterministic selections. The symbol at the top of the stack Γ: This is the symbol located at the highest position in the stack. This symbol is consistently removed from the stack during the transition prior to any new string being added. This "pop-then-push" process is essential for how the stack is handled. The codomain P(Q×Γ ∗) indicates that the transition may result in a collection of possible subsequent configurations. This collection illustrates the non-deterministic characteristics of PDAs, where numerous computational routes may be investigated concurrently. Every component in this collection is a tuple (q ,), where: q Q represents the subsequent state the PDA moves to. Γ ∗ is a sequence of stack symbols that substitutes the removed top symbol. The length and content of dictate the particular stack operation: If = (the empty string), the top symbol X is removed, and nothing is added back to the stack. This essentially

executes a clean pop operation. If is equal to 1

2 … j with j 1, the upper symbol X is removed, and then the sequence j … 2 1 is added to the stack. An essential convention is that is moved so that the first symbol of (reading from left to right) swaps places with the new top of the stack. For instance, if =XY, the character Y is pushed initially, then X. Therefore, X will be at the top of the stack. If =Z, then Z is pushed, which substitutes X with Z. If =X (meaning the symbol popped is pushed again), it's essentially a "peek at top of stack" action that doesn't alter the stack's total content but enables the PDA to respond to the top symbol without eliminating it. This adaptability in moving strings of symbols enables intricate stack operations, like pushing several markers for nested constructs or pushing a series of symbols that stand for a production rule in a context-free grammar. q 0 : The starting state. This is the specific, assigned state where the PDA starts before handling any input. All calculations of the PDA commence from this particular state, serving as the starting point for any string handling. q 0 Q Z 0 : The beginning stack symbol. This unique symbol is added to the stack prior to processing any input and before the execution of any other stack operations. It acts as an essential indicator for the base of the stack, functioning as a sentinel. Its existence stops the PDA from prematurely trying to pop from an empty stack (an "underflow" error) and, crucially, enables the PDA to recognize when the stack is "actually empty" (i.e., containing solely Z 0). This is especially important for acceptance empty stack, where the PDA must determine when all pertinent

The stack symbols have been removed. Z 0 Γ F: A limited collection of final or accepting states. If the PDA attains one of these states after completing the entire input string, the string is deemed accepted the final state. These states indicate a successful computation route, showing that the input string adheres to the language recognized the PDA according to its internal mechanisms and stack functions. F Q Configurations of a PDA A configuration (or instantaneous descrip-

tion, ID) of a PDA represents the entire state of the automaton at any specific moment in its computation. It's a snapshot that contains all the essential details required to resume the computation from that point. A configuration is formally defined as a triplet: (q,w,) where: q Q indicates the current state of the finite automaton. This indicates the current processing stage of the PDA. w $\Sigma *$ represents the input string that still needs to be processed. The PDA scans characters from the beginning of w. While the PDA processes input symbols, w gradually decreases in length. $\Gamma *$ represents the current stack content, with the top of the stack being the leftmost symbol. For instance, if =XYZ, then X occupies the highest position, Y is situated under X, and Z is at the lowest point (above Z is 0). The content of the stack evolves dynamically with every transition. The Yield Relation The transition function determines how the PDA shifts from one configuration to another in a single step. This action, symbolizing one computation step, is formally denoted the "yields" relation, represented as M (or just when the PDA M is evident from the context). If the transition function (q,a,X) includes the pair (q ,), then for any leftover input string w $\Sigma *$ and for any stack content $\Gamma *$: (q,aw,X) M (q ,w,) Let's analyze each component of this formal equation using a specific example. Assume there is a transition (q1, 'a', 'S') = (q2, 'AB'). This indicates: Present setup: (q 1 ,'a'w,'S') The PDA is in status q 1 . The upcoming symbol to be read is 'a'. The stack's peak is 'S', with denoting the remainder of the stack beneath 'S'. Following the configuration after a single move: (q 2 ,w,'AB') The PDA shifts to state q 2 . The input symbol 'a' is absorbed, resulting in w as the leftover input. (If it were an -transition, then a would equal , and the input string would still be aw). The uppermost stack symbol 'S' is removed. The stack receives the string 'AB'. According to the convention, 'B' is pushed first, followed 'A', making 'A' the new top of the stack. The remaining stack, , stays the same beneath the recently added 'AB'. The yield relation's reflexive and transitive closure

is represented M ∗ . This indicates a series of zero or more actions. If a series of zero or more transitions moves the PDA from an initial state C 1 to a final state C k , we denote this as C 1 M ∗ C k . This enables us to outline the complete computation trajectory of the PDA, from its beginning to any intermediate or final state, without specifying each individual step. (q 1 ,w 1 , 1) M ∗ (q k ,w k , k) Acceptance Conditions A string w Σ ∗ is accepted a PDA M if it meets either of two standard and equivalent acceptance criteria. The selection of acceptance criteria does not influence the category of languages that can be identified (context-free languages), yet one could be

more practical for certain PDA configurations or theoretical demonstrations. A string w is considered accepted a final state if, beginning from the initial configuration (q 0 ,w,Z 0), the PDA reaches a configuration where all input has been processed (in the input position), and the PDA is situated in an accepting state (a state belonging to the set F). At this stage, the stack's contents are irrelevant for acceptance under this condition; it may be empty or have any symbols. This condition intuitively indicates that the PDA has effectively "analyzed" the input and arrived at a recognized final state, similar to a Finite Automaton. For instance, a PDA created to identify valid arithmetic expressions may achieve a final state only after accurately processing all operators and operands, irrespective of the remaining items on the stack (such as intermediate results of calculations). This condition is frequently favored in real-world parsing applications, where achieving a final state indicates the successful identification of a complete program or statement. L(M)=w Σ ∗ (q 0 ,w,Z 0) M ∗ (q, ,) for some q F and Γ ∗ 2. A string w is accepted an empty stack if, beginning from the initial configuration (q 0 ,w,Z 0), the PDA can achieve a configuration where all input has been processed (in the input position), and the stack is entirely empty (in the stack content position). In this situation, the PDA's final state is irrelevant; any state q Q is permissible provided the stack is empty. This condition intuitively

signifies that every opening symbol has been paired with its respective closing symbol, and all necessary stack operations for the input have been executed, effectively achieving a "balanced" stack. A traditional instance is

identifying the language of matched parentheses, where every (is added and every) removes a (, and the string is accepted if the stack is clear at the conclusion. The stack will only be considered truly empty when the initial stack symbol Z 0 is also removed. N(M)=w $\Sigma *$ (q 0 ,w,Z 0) M $*$ (q, ,) for some q Q It is a widely accepted and essential result in automata theory that these two acceptance criteria are equivalent regarding the class of languages they can identify. Thus, for every language L accepted a PDA through final state, there is another PDA that accepts L via empty stack, and conversely. This equivalence streamlines the theoretical analysis of context-free languages, since one can opt for the more suitable acceptance condition for a particular proof or construction. For example, to change a PDA that accepts final state to one that accepts empty stack, a new initial stack symbol and a new final state can be introduced. From any original final state, the PDA moves to this new final state, removing all remaining stack symbols until only the new initial stack symbol remains, which is subsequently popped. An analogous structure is present for the opposite transformation. Non-Determinism in PDAs Pushdown Automata are fundamentally non-deterministic, in contrast to deterministic finite automata (DFAs). The transition function takes a specific (state, input, stack top) triplet and maps it to a set of potential next configurations, which may not be just one. This indicates that from one configuration, a PDA could have several legitimate moves, enabling it to "predict" the right route for acceptance. A string is accepted if there exists at least one sequence of actions that results in an accepting configuration. This non-determinism is not just a convenience; it is essential for identifying the entire class of context-free languages. Think about

the set of palindromes in the alphabet a,b, L pal =w a,b $*$ w=wr ,

where wr denotes the reverse of w. Instances comprise "aba", "abba", "abccba". To identify this type of language, a PDA must push symbols onto the stack until it arrives at the "center" of the string, and subsequently begin popping symbols to correspond with the rest of the input. The problem is that the PDA cannot determine the midpoint until it has moved beyond it. A non-deterministic PDA is able to "guess" the midpoint through an -transition that shifts its state from "pushing" to "matching" at any moment, and if it guesses accurately for an input palindrome, that route results in acceptance. A deterministic PDA (DPDA) cannot make such assumptions and would need to look ahead, which is not allowed. Likewise, examine the language L=a n b n n 0 a n b 2n n 0. A PDA that recognizes this language must determine, after processing a series of 'a's, whether to pair them with an equal count of 'b's or double that amount of 'b's. This choice should be made without considering the 'b's that come afterward. A non-deterministic PDA may have multiple transitions from a state upon reading 'a's: one directing to a path that corresponds to a n b n and another directing to a path that corresponds to a n b 2n. If the input string is part of any segment of the union, one of these non-deterministic paths will be successful. A DPDA is unable to manage this ambiguity without extra details. This differentiation emphasizes that deterministic pushdown automata (DPDAs) are a strictly weaker category of machines compared to non-deterministic pushdown automata (NPDAs); DPDAs cannot recognize every context-free language. The languages accepted DPDAs are referred to as deterministic context-free languages, which constitute a proper subset of the context-free languages. This holds considerable importance.

practical considerations in parsing: although every context-free language can be recognized NPDAs, not every one can be parsed efficiently deterministic parsers (such as LR parsers, which rely on DPDAs). Grammars resulting in non-deterministic PDAs are frequently referred to as ambiguous grammars, and addressing this ambiguity is

an essential objective in the design of compilers. The formal definition of a PDA offers a clear mathematical structure for comprehending these robust computational models and their essential function in the Chomsky hierarchy, connecting regular languages (recognized FAs) with more intricate language forms, ultimately supporting the theoretical bases of programming language development and compilation.

Chunck 3

Sjálfvirkar ýtingarvélar og Turing vélar

Teóríunni um tölvunarfræði er einkennd af mismunandi abstrakt tölvunarvélum, hver einasta var hönnuð til að skýra út ákveðna flokkun reiknigetu og til að afmarka mörkin á tilteknum tungumálafjölskyldum. Among the most essential of these elements, which act as fundamental pillars for the whole structure of computability theory, are Turing Machines (TMs) and Pushdown Automata (PDAs). Though both entities function as foundational models of computation, their inherent memory structures and operational frameworks give rise to significant differences in the languages that can be effectively recognized them. This presentation will formally compare these two unique computational models, highlighting their key features, their individual functioning methods, and their specific rankings within the framework referred to as the Chomsky hierarchy, thus shedding light on the progressive nature of computational capability. A Pushdown Automaton (PDA) is formally specified as a 7-tuple, represented as M PDA =(Q,Σ,Γ, ,q 0 ,Z 0 ,F). In this formal specification, every component has a specific functional meaning. Q represents a finite, non-empty

collection of internal states, symbolizing the distinct internal arrangements of the automaton's finite control unit. Σ denotes the finite, non-empty input alphabet, consisting of the symbols that are

read from the input tape in a sequential manner. Γ represents the finite, non-empty stack alphabet, including all symbols that can be manipulated within the auxiliary memory structure. indicates the transition function, serving as the core mechanism of the automaton, defining its behavior. q 0 indicates the singular initial state, where all computational activities begin. Z 0 denotes the starting stack symbol, a sentinel added to the stack before any processing begins. Finally, F includes the finite collection of final or accepting states, achieving which indicates successful identification of an input string. The key defining feature of a PDA lies in its stack, which operates as a Last-In, First-Out (LIFO) memory system. This LIFO method means that the most recently added symbol to the stack is always the first one to be removed, making it highly effective for handling nested dependencies, like matching parentheses or the organizational layout of programming constructs. A configuration, or instantaneous description (ID), of a PDA is accurately represented a triplet (q,w,), where q Q indicates the current state of the finite control, w $\Sigma *$ denotes the part of the input string remaining to be processed, and $\Gamma *$ reflects the current contents of the stack, with the leftmost symbol typically signifying the topmost element. The transition function, , is formally defined as a mapping from the Cartesian product $Q\times(\Sigma)\times\Gamma$ to the power set $P(Q\times\Gamma *)$. This mapping clearly shows that a computational action is based on the existing state, the input symbol currently under review (or the empty string , denoting an -transition that requires no input), and the symbol located at the top of the stack. This action requires the use of an input symbol (unless it's), the elimination (popping) of the highest stack symbol, and the later inclusion (pushing).

of a sequence of characters onto the stack, which can consist of the empty string, thus effectively executing a pure pop. The formal

representation of the yield relation for a PDA, illustrating one computational step, is as follows: (q,aw,X) PDA (q ,w,) if (q ,) (q,a,X). In this case, (q,aw,X) indicates the configuration before the transition, where q signifies the current state, a is the input symbol (or) being read, X is the top symbol on the stack, and w and represent the remaining input and stack, respectively. The derived configuration (q ,w,) reflects the updated state q , the remaining input w after a has been handled, and the stack content , where denotes the string that was added to the stack after X was removed. The reflexive and transitive closure of this relation, PDA $*$, represents a series of zero or more computational actions, allowing for the examination of the entire processing of an input string. PDAs are widely recognized as the computational framework for context-free languages (CFLs), a category of languages crucial for the accurate representation of programming language syntax, encompassing nested constructs such as if-else statements and function invocations. It's crucial to understand that non-determinism in PDAs is not just a supplementary trait but a vital attribute that greatly enhances their expressive capability; the category of languages accepted non-deterministic PDAs (NPDAs) is clearly and strictly a superset of those accepted deterministic PDAs (DPDAs). This essential difference is clearly illustrated languages like palindromes (e.g., L=w a,b $*$ w=w R), the identification of which fundamentally requires non-deterministic decisions for effective handling, since the reversal point in the string cannot be deterministically determined without infinite lookahead. Such an automaton could non-deterministically choose to move from a

Transitioning from a "pushing" phase to a "matching" phase at any moment, and if the right selection is made for a specific input, the string is accepted. This idea is extensively examined in academic publications like [arXiv:1608.02640], which clarifies the constraints of DPDAs. The practical outcome of this non-determinism is that although all CFLs can be parsed, not every one can be efficiently parsed deterministic

top-down or bottom-up algorithms without alterations or lookahead. In contrast, a Turing Machine (TM) embodies a computational framework of considerably higher capability, acting as the theoretical model for universal computation. It is officially represented as a 7-tuple, M TM =(Q,Σ,Γ, ,q 0 ,B,F). In this expression, Q represents a limited collection of states, similar to the PDA's internal regulation. Σ denotes the input alphabet, which is the collection of symbols originally found on the tape. Γ represents the tape alphabet, a broader collection of symbols that encompasses Σ and the unique blank symbol B (Σ Γ). refers to the transition function, which dictates the behavior of the TM. q 0 denotes the starting state. B Γ Σ denotes the distinct blank symbol that fills every unoccupied infinite section of the tape not taken up the input. F represents the collection of terminal states. The key feature of a TM lies in its memory structure: an unlimited, theoretically endless tape that allows for random access, facilitating actions like reading, writing, and two-way movement of the tape head. In contrast to the LIFO limitation of the stack, the tape head of the TM can traverse left or right throughout the tape, enabling information to be stored, accessed, and altered at any position. This ability for random access is the origin of the TM's great strength. A TM's configuration is typically expressed as (q, 1 X 2), specifically indicating the current state q and the complete contents of the tape .

1 X 2 (where 1 denotes the non-blank content to the left of the head, X represents the symbol located beneath the head, and 2 signifies the non-blank content to the right), with the tape head situated directly above the symbol X. The transition function associates Q×Γ with Q×Γ×L,R, suggesting that a computational action depends on the current state and the symbol currently under the tape head. This action leads to a change in state, the addition of a new symbol on the tape at the existing head position, and a movement of the tape head in one direction (Left or Right). The yield relation for a TM, outlining a single transition, can be formally stated as: (q, 1 X 2

) TM (q , 1 Y 2) if (q,X)=(q ,Y,R) and 2 is 2 with its first symbol transferred to 1 ; (q, 1 X 2) TM (q , 1 Y 2) if (q,X)=(q ,Y,L) and 1 is 1 with its last symbol moved to 2 . The computational power of Turing Machines clearly exceeds that of Pushdown Automata a significant margin. TMs can identify recursively enumerable languages (RELs), which are also referred to as Turing-recognizable languages. This broad category of languages includes all context-free languages and also covers a wide range of others, particularly context-sensitive languages (CSLs) and even issues that cannot be resolved simpler models. The connection among these different language categories is defined a rigid ranking, commonly known as the Chomsky Hierarchy: L Regular L CFL L CSL L REL. This means that all regular languages are context-free, all context-free languages are context-sensitive, and all context-sensitive languages are recursively enumerable, with each inclusion being distinct. Moreover, the esteemed

The Church-Turing Thesis claims that any problem suitable for algorithmic resolution can, in theory, be addressed a Turing Machine, thus positioning TMs as the theoretical standard for the notion of computability and the final theoretical boundary of what is computable. Unlike PDAs, the existence of non-determinism does not enhance the intrinsic computational strength of Turing Machines; non-deterministic TMs (NTMs) are demonstrably equivalent in computational power to deterministic TMs (DTMs). This equivalence is proven through a constructive demonstration that any NTM can be replicated a DTM, though it may entail a considerable rise in computation duration, as evidenced scholarly studies like those found in [arXiv:2009.04342]. This suggests that for TMs, non-determinism does not provide extra computational capabilities, but rather may influence the efficiency of the computation process. In summary, although both Pushdown Automata and Turing Machines are essential foundational models in theoretical computation, their differing memory structures are the critical ele-

ments that determine their unique computational abilities and, therefore, the types of languages they can recognize. Pushdown Automata, restricted their stack-oriented memory, are fundamentally confined to recognizing context-free languages, thus aiding in the handling of nested structural forms. Their non-determinism is an essential characteristic for their expressive capabilities. In contrast, Turing Machines, equipped with an infinitely accessible tape, have the ability to identify any problem that can be solved algorithmically, there embodying the highest theoretical boundary of computation. The TM's capability to read, write, and traverse freely along an infinite tape provides it with universal computational strength, a feature that is fundamentally lacking in the more limited PDA. The conceptual landscape of theoretical computer science is marked the existence of numerous abstract computational devices, each carefully crafted to define a particular category of computational capability and to clarify

the limits of specific language groups. Among the most essential of these concepts, which act as fundamental pillars for the whole structure of computability theory, are Turing Machines (TMs) and Pushdown Automata (PDAs). These models are more than just theoretical interests; they offer a solid mathematical foundation for grasping the strengths and weaknesses of computation itself, serving as the basis for disciplines like compiler design, algorithm analysis, and artificial intelligence. Although both constructs act as foundational models of computation, their inherent memory structures and operational frameworks create significant differences in the types of languages each can recognize. This presentation will systematically compare these two separate computational models, highlighting their key features, their unique functioning processes, and their exact positions within the framework referred to as the Chomsky hierarchy, thus clarifying the progressive nature of computational capabilities and the escalating intricacy of languages that can be formally articulated and identified. A Pushdown Automaton (PDA) is formally characterized as a 7-tuple, represented

as M PDA =(Q,Σ,Γ, ,q 0 ,Z 0 ,F). In this formal specification, every component holds a specific functional importance, aiding in the overall behavior and computational capabilities of the automaton. Q is a finite, non-empty collection of internal states that represent the distinct internal arrangements of the automaton's finite control unit. These states represent the ongoing stage of computation, including whether the automaton is parsing an opening bracket, anticipating a closing one, or has faced an error condition. Σ denotes the finite, non-empty input alphabet, consisting of the symbols that are read in sequence from the input tape. This alphabet specifies the allowable characters that create the strings of the language the PDA aims to identify, similar to the lexical tokens in a programming language. Γ ,

,

can be altered concerning the supplementary memory framework. It is important to differentiate Γ from Σ, as Γ frequently contains specialized indicators (e.g., for subroutine invocations or block boundaries) that are not included in the input language but are vital for the PDA's internal management. represents the transition function, which is the core of the automaton, directing its actions outlining how the PDA transitions between states. q 0 indicates the singular initial state, from which all computation procedures begin, setting the automaton's starting location for any specific input string. Z 0 denotes the starting stack symbol, a sentinel positioned on the stack before any operations begin. This symbol acts as an essential marker for the bottom of the stack, helping to avoid underflow errors and enabling the PDA to recognize when its stack is truly empty, a state that is frequently vital for language acceptance. Finally, F includes the limited set of final or accepting states, reaching which indicates successful recognition of an input string, depending on the selected acceptance criterion. The most essential distinguishing feature of a PDA lies in its stack, which operates as a Last-In, First-Out (LIFO) memory configuration. This LIFO principle indicates that the final symbol added to

the stack will always be the first to be removed, making it especially effective at handling nested dependencies, such as matching parentheses in mathematical formulas, the structured hierarchy of programming constructs (e.g., nested begin-end statements or function calls), or the alignment of XML tags. This memory architecture grants the PDA an unlimited, albeit limited, ability to "retain" any levels of nested data. A configuration, or instantaneous description (ID), of a PDA is accurately represented the triplet (q,w,), where q Q signifies the present state of the finite control, w $\Sigma *$ indicates the portion of the input string that remains to be processed.

to be processed, and $\Gamma *$ denotes the present contents of the stack, with the leftmost symbol typically indicating the top element. The transition function, , is rigorously defined from the Cartesian product $Q\times(\Sigma)\times\Gamma$ to the power set $P(Q\times\Gamma *)$. This mapping clearly shows that a computational step depends on the current state, the input symbol currently under review (or the empty string , representing an -transition that utilizes no input), and the symbol located at the top of the stack. An -transition is especially potent because it permits the PDA to alter its stack or switch states without moving its read head on the input tape, facilitating intricate internal adjustments or non-deterministic "assumptions" concerning the input configuration. This action requires the use of an input symbol (unless it's), the elimination (popping) of the uppermost stack symbol, and then the addition (pushing) of a sequence of symbols onto the stack, which can also be the empty string, thus effectively executing a pure pop. For example, if (q 1 ,'a','X')=(q 2 ,'YZ'), then when reading 'a' in state q 1 with 'X' at the stack's top, the PDA moves to state q 2 , removes 'X', and adds 'YZ' (with 'Z' pushed first, followed 'Y', placing 'Y' at the new top). The formal representation of the yield relationship for a PDA, indicating one computational step, is defined as follows: (q,aw,X) PDA (q ,w,) if (q ,) (q,a,X). In this context, (q,aw,X) signifies the configuration before the transition occurs, where q is the

existing state, a is the input symbol (or) that will be read, X is the character at the top of the stack, and w and represent the remaining input and stack, respectively. The resulting configuration (q ,w,) signifies the updated state q , the

input w left after processing a, along with the stack content , where represents the string added to the stack after X was removed. The reflexive and transitive closure of this relation, PDA $*$, outlines a sequence of zero or more computational steps, allowing for the examination of the processing of an entire input string from an initial configuration (q 0 ,w,Z 0) to a final accepting state. PDAs are widely recognized as the computational model for context-free languages (CFLs), a category of languages essential for accurately defining programming language syntax, encompassing nested constructs such as if-else statements, for loops, and function invocations. The stack's capability to keep track of unmatched opening delimiters (e.g., parentheses, braces) and guarantee their correct closure is essential for parsing these languages. It is crucial to recognize that non-determinism in PDAs is not just an added feature but a fundamental trait that greatly enhances their expressive capability; the category of languages accepted non-deterministic PDAs (NPDAs) is clearly and distinctly a superset of those accepted deterministic PDAs (DPDAs). This basic difference is clearly illustrated languages like palindromes (e.g., L=w a,b $*$ w=w R), whose recognition inherently requires non-deterministic options for accurate handling. A PDA that identifies palindromes needs to push symbols onto the stack for the initial half of the string and subsequently pop them for the latter half. The exact "center" of the string, where the shift from pushing to popping must take place, cannot be definitively pinpointed without limitless lookahead. As a result, this automaton could non-deterministically choose to shift from a "pushing" phase to a "matching" phase at any moment, and if the right selection is made for a specific input, the string is accepted. An additional example is the language L =a n

b$^n n0a^n b(2n)n0.APDAneedstodetermine, afterprocessingaseriesof'a's, ifitsh$
$deterministicPDAcandivergeintotworoutes : onetryingtomatcha^n b^n andtheothero$
$deterministicrouteswillsucceed.Thisideaisdeeplyexaminedinacademicliteraturel$
1608.02640], $whichclarifiestheconstraintsofDPDAsandtherequirementofnon-$
$determinismfortheclassofCFLs.Thepracticalconsequenceofthisnon-$
$determinismisthat, althoughallCFLscanbeparsed, noteveryonecanbeparsedeffic$
$down(e.g., LL(k))orbottom-up(e.g., LR(k))parsingmethodswithoutalterationsor$
$free, ortheapplicationofmoreintricateparsingmethodsforambiguousornon-$
$deterministiccontext-freelanguages.Ontheotherhand, aTuringMachine(TM)e$
$purposecomputation.Itisofficiallycharacterizedasa7-tuple, MTM =$
$(Q, , , , q0, B, F).Inthisformulation, Qrepresentsafinitesetofstatessimilartothe$

0 represents the starting condition. B Γ Σ signifies the distinct blank symbol that occupies all endless sections of the tape not used the input, establishing a definitive limit for the input string and an unlimited workspace. F represents the collection of final states, where entering signifies the acceptance of the input string. The primary feature of a TM lies in its memory structure: an unbounded, theoretically endless tape that allows random access, facilitating operations for reading, writing, and two-way movement of the tape head. In contrast to the stack's LIFO limitation, the tape head of the TM can traverse left or right across any segment of the tape, enabling information to be stored, fetched, and altered at any location. The ability for random access, combined with the tape's limitless capacity, is the core reason for the TM's remarkable strength, allowing it to emulate any other computational model and execute any imaginable algorithm. A TM's configuration is typically expressed as (q, 1 X 2), which clearly indicates the current state q, the complete tape content 1 X 2 (where 1 represents the non-blank content to the left of the head, X is the symbol beneath the head, and 2 signifies the non-blank content to the right), while the head is situated directly above the symbol X. The underscore underneath X distinctly marks the location of the head. The transition function connects Q×Γ to Q×Γ×L,R, indicating that

a computational step depends on the current state and the symbol currently beneath the tape head. This action leads to a change to a different state, the inscription of a new symbol on the tape at the current head location, and a directional shift of the tape head (Left or Right). For example, if (q 1 ,'a')=(q

2 ,'b',R), then in state q 1 with 'a' under the head, the TM transitions to state q 2 , writes 'b' over 'a', and shifts its head one position to the right. The yield relation for a TM, explaining a single transition, can be expressed as follows: (q, 1 X 2) TM (q , 1 Y 2)if (q,X)=(q ,Y,R) and 2 is 2 with the first symbol shifted to 1 (q, 1 X 2) TM (q , 1 Y 2)if (q,X)=(q ,Y,L) and 1 is 1 with the last symbol shifted to 2 The reflexive and transitive closure of this relation, TM ∗ , facilitates the description of a complete computation, from the starting configuration to a halting (accepting or rejecting) configuration. Turing Machines' computational power clearly exceeds that of Pushdown Automata to a significant extent. TMs can identify recursively enumerable languages (RELs), referred to as Turing-recognizable languages. This broad category of languages includes all context-free languages and also covers a wide range of others, particularly context-sensitive languages (CSLs) and even issues that are undecidable more straightforward models, like the Halting Problem. The connection among these different language classes is marked a definitive hierarchy, commonly known as the Chomsky Hierarchy: L Regular L CFL L CSL L REL. This means that all regular languages are also context-free, all context-free languages are context-sensitive, and all context-sensitive languages are recursively enumerable, with each inclusion being proper. For instance, L Regular comprises languages such as a ∗ b ∗ , L CFL contains a n b n , L CSL consists of a n b n c n , and L REL encompasses the collection of all valid.

Descriptions of Turing Machines. Moreover, the esteemed Church-Turing Thesis asserts that any problem suitable for an algorithmic solution can, in theory, be addressed a Turing Machine, thus position-

ing TMs as the theoretical standard for the notion of computability and the definitive theoretical boundary of what can be computed any effective method. This thesis, although not a mathematical theorem, is a broadly recognized fundamental concept in computer science. Unlike PDAs, the presence of non-determinism does not enhance the fundamental computational strength of Turing Machines; non-deterministic TMs (NTMs) are demonstrably equal in computational capability to deterministic TMs (DTMs). This equivalence is shown through a constructive proof that any NTM can be replicated a DTM, though it may result in a considerable rise in computation time (for instance, an exponential delay in certain instances), as supported scholarly work like that found in [arXiv:2009.04342]. This suggests that for Turing Machines, non-determinism does not provide extra capability regarding what is computable (recognizability), only possibly how efficiently it can be computed (complexity). In summary, although both Pushdown Automata and Turing Machines serve as crucial foundational models in theoretical computation, their differing memory structures are the key elements that determine their particular computational abilities and, therefore, the types of languages they can identify. Pushdown Automata, limited their stack memory, can only recognize context-free languages, which aids in processing nested structure patterns. Their non-determinism is an essential attribute for their expressive capabilities, allowing for the identification of the entire category of CFLs. In contrast, Turing Machines, equipped with an infinite, randomly accessible tape, have the ability to identify any problem that can be solved algorithmically, there embodying the supreme theoretical boundary of computation. The TM's capability to read, write, and traverse freely across an infinite tape provides

It possesses universal computational ability, a feature that is essentially lacking in the more limited PDA. This key distinction in memory access and capacity is the crucial factor setting apart the computational abilities of these two essential abstract machines.

Chunck 4

Kynning á niðurdrepandi uppteknum bever

(fráhrindandi froski)

$$\sigma : \{x \mid x \text{ is pushdown automata}\} \to \mathbb{N}$$

$$\text{PDA} \mapsto \sigma(\text{PDA})$$

$$\sigma(\text{PDA}) = \max\{t \in \mathbb{N} \mid \exists w \in \Sigma^* \text{ so that } M \text{ halts at the input } w \text{ on exactly } t \text{ steps}\}$$

$$\text{pudau} : \mathbb{N} \to \mathbb{N}$$

$$n \mapsto \text{pudau}(n)$$

$$\text{pudau}(n) = \max\{\sigma(\text{PDA})$$

$$\mid \text{PDA is an n-state 2-symbol pushdown automaton that halts}\}$$

1. The field of theoretical computer science is significantly influenced the examination of automata, which are abstract computational models offering a solid foundation for grasping the boundaries and functionalities of computation. From the most basic Finite State Automata (FSA) to the universally capable Turing Machines (TMs), every model presents unique memory mechanisms that determine its computational strength and, importantly, the decidability of its characteristics. This chuck investigates the upper limits of halting times for Pushdown Automata (PDAs), a distinctive class of machines in this hierarchy, presenting and examining two new functions: (PDA) and pudau(n). These functions seek to measure the longest potential computation paths for PDAs, creating an intriguing parallel with and a significant contrast to the established Busy Beaver function for Turing Machines. 1.1 Background in Computability Theory and Automata Computability theory, a fundamental aspect of theoretical computer science and mathematical logic, investigates the essential inquiries regarding what is computable and what is not. At its core is the idea of an algorithm and the structured models that represent it. Automata theory, as a subfield, offers a structured categorization of these computational models according to their memory functions and the types of formal languages they can identify.

of a sequence of characters onto the stack, which can consist of the empty string, thus effectively executing a pure pop. The formal representation of the yield relation for a PDA, illustrating one computational step, is as follows: (q,aw,X) PDA (q ,w,) if (q ,) (q,a,X). In this case, (q,aw,X) indicates the configuration before the transition, where q signifies the current state, a is the input symbol (or) being read, X is the top symbol on the stack, and w and represent the remaining input and stack, respectively. The derived configuration (q ,w,) reflects the updated state q , the remaining input w after a has been handled, and the stack content , where denotes the string that was added to the stack after X was removed. The reflexive

and transitive closure of this relation, PDA $*$, represents a series of zero or more computational actions, allowing for the examination of the entire processing of an input string. PDAs are widely recognized as the computational framework for context-free languages (CFLs), a category of languages crucial for the accurate representation of programming language syntax, encompassing nested constructs such as if-else statements and function invocations. It's crucial to understand that non-determinism in PDAs is not just a supplementary trait but a vital attribute that greatly enhances their expressive capability; the category of languages accepted non-deterministic PDAs (NPDAs) is clearly and strictly a superset of those accepted deterministic PDAs (DPDAs). This essential difference is clearly illustrated languages like palindromes (e.g., L=w a,b $*$ w=w R), the identification of which fundamentally requires non-deterministic decisions for effective handling, since the reversal point in the string cannot be deterministically determined without infinite lookahead. Such an automaton could non-deterministically choose to move from a

At the bottom level of this computational hierarchy are Finite State Automata (FSAs). These devices have a limited number of states and no additional memory. Their processing capability is confined to identifying regular languages, which are defined straightforward, repetitive patterns. The actions of an FSA are completely dictated its present state and the input symbol it processes. Moving up the hierarchy, Pushdown Automata (PDAs) signify an important advancement in computational ability. In automata theory, PDAs are considered one of the essential structures, defined finite-state controllers enhanced with a unique memory component known as a stack. This stack functions based on a First-In, Last-Out (FILO/LIFO) principle, indicating that the most recently added item is the first to be taken out. This unlimited stack memory enables PDAs to identify context-free languages (CFLs), a wider category of languages than regular languages, which includes structures needing memory beyond finite states, such as recur-

sive or nested patterns like balanced parentheses. The capacity to push and pop symbols from the stack allows a PDA to ”retain” an unlimited number of previous input symbols, a feature lacking in FSAs. At the top of the classical computational hierarchy sit Turing Machines (TMs). TMs possess an endless tape that functions as both input and storage, enabling unlimited reading, writing, and motion actions along the tape. This versatile memory enables TMs to identify recursively enumerable languages and replicate any algorithm runnable on a contemporary computer, establishing the foundation of the Church-Turing thesis. The established hierarchy of computational models—FSA $<$ PDA $<$ TM—demonstrates that each successive model offers greater computational power and memory adaptability. The hierarchical structure of these computational models is essential for comprehending the relative strengths and weaknesses of PDAs. This hierarchy suggests that issues resolvable are a strict superset of the problems that Finite State Automata can solve but a strict subset of those that Turing Machines can tackle. This difference in computational strength is not just regarding the category of languages recognized (regular, context-free, recursively enumerable) but significantly affects the decidability of essential characteristics, like halting. The limited memory access of a PDA (LIFO stack) versus a TM's random-access tape is the reason for this difference, which will be crucial to the computability of pudau(n). The fundamental restriction of PDAs, particularly their stack-based memory, is the primary cause that certain issues, such as the halting problem, are decidable for PDAs, in contrast to TMs. This key distinction will serve as the basis for the claim that pudau(n) is a computable function, clearly setting it apart from the non-computable Busy Beaver function. 1.2 The Halting Problem and Its Significance in Computation Constraints At the core of computability research and the theoretical boundaries of algorithms lies the Halting Problem. This fundamental issue inquires if it's feasible to identify, based on a description of any given computer program and a specific input, whether

that program will ultimately stop executing (halt) or persist in running indefinitely. In 1936, Alan Turing famously demonstrated that the Halting Problem is undecidable for Turing Machines. This implies that there cannot be a universal algorithm that accurately resolves the Halting Problem for every potential program-input combination. This undecidability creates a core limitation in forecasting any program's behavior, significantly influencing the definition and computability of functions that measure maximum computational duration. The undecidability of the Halting Problem is clearly demonstrated through the Busy Beaver function, S(n). Introduced Tibor Radó in 1962, S(n) measures the highest number of steps an n-state, 2-symbol Turing machine can execute prior to halting when initiated on a tape filled with all 0s (blank). The non-computability of S(n)

is a direct result of the undecidability of the Halting Problem for TMs; if S(n) could be computed, it would allow for the resolution of the Halting Problem, which is recognized as unachievable. As a result, the Busy Beaver function increases asymptotically at a rate surpassing any computable function, achieving unimaginably large values for small n, for example, S(5)=47,176,870 and S(6)>$10\uparrow\uparrow15$ (a tower of 15 tens). The definition of (PDA) and pudau(n) as functions measuring maximum halting times for PDAs inherently requires a conversation about the decidability of the Halting Problem for this particular class of automata. If the Halting Problem for PDAs were undecidable, (PDA) would lack definition for any PDA that fails to halt on any input (since its maximum would be infinite). Thus, pudau(n), which is defined as the highest value of (PDA) across a collection of halting PDAs, would also be non-computable. This prompt departure from the Busy Beaver function is a key aspect that will be examined thoroughly in this document. The definition of pudau(n) as a mapping to N (natural numbers) indicates that the highest halting time for any halting PDA is finite, and that we can identify the collection of halting PDAs (those that halt for at least one input). This can happen only if the Halting Prob-

lem for PDAs is solvable. This fundamental decidability is the key attribute that sets pudau(n) apart from S(n) and will act as a central focus, emphasizing the distinctive role of PDAs within the hierarchy of computational models. 1.3 Overview of Pushdown Automata (PDAs) Pushdown Automata represent a category of theoretical machines that enhance the abilities of Finite State Automata through the addition of an unrestricted stack memory. This extra memory is the key feature that allows PDAs to identify context-free languages (CFLs), a category of languages that encompasses structures needing memory.

beyond limited states, like correctly nested parentheses or expressions. The stack functions on a Last-In, First-Out (LIFO) principle, indicating that data is saved and retrieved in a particular order: the most recent symbol added to the stack is the first one ready to be removed. A non-deterministic Pushdown Automaton (NPDA) is formally defined as a 7-tuple M=(Q,Σ,Γ, ,q 0 ,Z 0 ,F), where: Q represents a finite collection of states. Σ . Γ . :Q×(Σ)×Γ→P(Q×Γ ∗) . This function governs the PDA's operations: for a current state, an input symbol (or for a transition with no input), and the symbol on top of the stack, it specifies a set of potential next states and strings to be added to the stack. The symbol P(X) represents the powerset of X, showing that may correspond to several next configurations, hence enabling non-determinism. The initial state is denoted q 0 Q. Z 0 Γ serves as the initial stack symbol, located at the base of the stack when the computation starts. F Q es el conjunto de estados de aceptación (finales). A transition expressed as (q,a,A) (p,) indicates that when the automaton is in state q, processes the input symbol a (or a= for an -move), and A is the symbol at the top of the stack, it can move to state p, remove A from the stack, and place the string Γ ∗ onto the stack (where the leftmost character of becomes the new top). The symbol in the input section of the transition function enables -moves or -transitions, permitting the machine to advance without reading

any input symbol. The symbol in the stack section of the transition indicates "do nothing" (i.e., neither pop nor push), which enables the PDA to process an input symbol while

disregarding the stack, or to add a symbol without removing, or to remove a symbol without adding. PDAs may be classified as either deterministic (DPDA) or non-deterministic (NPDA). Although non-deterministic pushdown automata (1npda's) define the entire set of context-free languages, deterministic pushdown automata (1dpda's) identify a narrower category referred to as deterministic context-free languages. Non-determinism is a fundamental characteristic of several significant PDA examples, enabling the machine to "predict" the correct course in a computation, such as estimating the midpoint of a string to verify palindromes. The formalisms for PDAs may differ, such as whether endmarkers exist on the input tape, yet these differences are typically viewed as "equivalent" in computational strength for fundamental pushdown automata. The stack's LIFO characteristic, although offering unlimited memory, enforces structural limitations that significantly restrict a PDA's computational abilities when compared to a Turing Machine's tape with random access. This limitation is exactly what renders the halting problem solvable for PDAs, in contrast to TMs, and therefore makes pudau(n) a computable function. The PDA's limitation in simulating a universal Turing machine, caused its stack restrictions, enables its halting problem to be decidable. This differentiation forms the foundation for the computability of pudau(n), establishing a distinct line between the "Busy Beaver" functions for PDAs and TMs. In contrast to Turing machines that can alter any section of their infinite tape at any moment, a PDA is limited to interacting exclusively with the top of its stack, significantly constraining its capacity to handle arbitrary data structures or replicate intricate recursive processes beyond context-free languages. This limitation on memory access is the reason for the variance in computational capability and, importantly, the solvability of the halting

problem for PDAs. 1.4 Overview of the and pudau Functions The exploration of maximum computation durations is an intriguing field.

within theoretical computer science, expanding the limits of understanding regarding the behavior of abstract machines. Building on the core principles of automata theory and computability, this chuck presents and thoroughly examines two functions aimed at measuring the maximum halting times specifically for Pushdown Automata. The functions (PDA) and pudau(n) are motivated the essence of the Busy Beaver problem for Turing Machines, yet their character is heavily influenced the unique computational traits of PDAs. The initial function, (PDA), measures the highest number of steps that a particular Pushdown Automaton can execute prior to stopping. It is defined as: $[\sigma : M \mid M \text{ is a pushdown automaton} \to \mathbb{N}][M \mapsto \sigma(M)].Thisdefinitionindicatesthat$ $[\text{pudau} : \mathbb{N} \to \mathbb{N}][n \mapsto \text{pudau}(n)]Inthiscontext, ndenotesthecountofstatesinthePu$ $symbol"constraintpertainstotheinputalphabet'ssize, akintotheconventionaldefin$

of the Busy Beaver function for Turing Machines, which generally employs a binary alphabet. The phrase "that halts" in the definition of pudau(n) is essential; it limits the collection of PDAs being analyzed to solely those that are ensured to halt on at least one input string. This guarantees that (PDA) is limited for every PDA in the collection, and consequently, pudau(n) is also limited. To situate these functions, it is crucial to remember the traditional Busy Beaver function, S(n), concerning Turing Machines. As mentioned earlier, S(n) is determined as the highest number of steps that an n-state, 2-symbol Turing machine performs before stopping when initiated on a tape filled with zeros. A key characteristic of S(n) is its non-computability, indicating that no algorithm can determine its values for every n. This non-computability stems from the Halting Problem's undecidability for Turing Machines, causing S(n) to increase more rapidly than any computable function. The clear definition of pudau(n) as a "Busy Beaver analogue" for PDAs instantly creates a comparative analytical structure. The key implication is that although both functions seek to optimize halting time

concerning a size parameter, the essential distinction in the underlying computational models (PDA versus TM) will result in markedly different properties of computability and rates of growth. This comparison will be crucial for grasping the inherent nature of pudau(n). The decidability of PDA halting (to be demonstrated in Section 2.2) implies that pudau(n) is a computable function, which sharply contrasts with S(n). This chuck will thoroughly examine this distinction, showcasing how memory architecture determines the boundaries of optimal computation. The difference between the infinite tape memory of a Turing Machine and the LIFO stack memory of a Pushdown Automaton is not simply a technical nuance; it is the fundamental characteristic that divides the domain of

distinguishing undecidability from decidability in this scenario, and thus, the computability of pudau(n) from the uncomputability of S(n). Formal Definition and Operational Semantics of Pushdown Automata To effectively examine the and pudau functions, a clear grasp of Pushdown Automata and their operational semantics is essential. This part presents a formal definition of a non-probabilistic Pushdown Automaton, outlines its elements, and explains the process of its computation and termination. 2.1 Elements of a Pushdown Automaton A Pushdown Automaton (PDA) is precisely characterized as a 7-tuple M=(Q,Σ,Γ, ,q 0 ,Z 0 ,F). Every component has a distinct function in determining the behavior and abilities of the automaton: Q (Finite Set of States): This represents a limited, non-empty collection of internal states that the PDA may occupy at any moment. These states signify the ”control unit” of the automaton, akin to states in a Finite State Automaton. For a PDA with n states, Q =n. Σ (Input Alphabet): This is a finite, non-empty collection of symbols that can be interpreted from the input tape. In the case of pudau(n), the input alphabet is limited to 2 symbols, meaning Σ =2. Γ (Σ Alphabet): A , . In contrast to the input alphabet, the definition of pudau(n) imposes no specific limitations on the size

of the stack alphabet, other than it being finite. Generally, Γ has a unique marker for the bottom of the stack (Z 0) along with additional symbols utilized for calculations. The selection of Γ can greatly affect the behavior and complexity of a PDA. (Transition Function): This is central to the PDA's operational mechanics, specifying how the automaton transitions from one configuration to another. It is a function :Q×(Σ)×Γ→P(Q×Γ∗). The domain of is made up of a triple: (current state, input symbol (or), symbol at the top of the stack). The set of output values

P(Q×Γ ∗) denotes the set of all pairs (next state, string to add to the stack). The employment of the power set shows that PDAs are fundamentally non-deterministic; from any specific configuration, numerous transitions could be feasible. A transition rule (q,a,A) (p,) indicates that if the PDA is in state q, the next symbol to read is a (or if no input is processed), and the top stack symbol is A, then the PDA can move to state p, remove A from the stack, and add the string to the stack. The string is pushed so that its leftmost character becomes the updated top of the stack. Specific scenarios for : If = (empty string), after popping A, no new element is added to the stack. This simply results in the removal of A. If =A, the top of the stack still is A, resulting in no alteration to the stack's contents. If =BA (with B as a new symbol), A is removed, and subsequently B and A are added, there substituting A with BA. If =B 1 B 2 …B k , A is removed, followed pushing B k ,…,B 2 ,B 1 in that sequence, making B 1 the top element. The -transitions (or -moves) are essential, enabling the PDA to alter its state and adjust its stack without using an input symbol. This ability can result in intricate and possibly lengthy computation paths. q 0 (Starting State): This is the distinct state from Q where the PDA initiates its computation. Z 0 (Starting Stack Symbol): This is the distinct symbol from Γ that is first positioned on the stack at the outset of the computation. It acts as a marker at the bottom of the stack. F (Subset of Accepting States): This is a portion of Q. If

the PDA

completes processing its input and reaches a condition in F, the input string is deemed accepted. One typical method to define acceptance is through an empty stack, which is equivalent for non-deterministic PDAs. 2.2 Configurations and Halting Conditions The computation of a PDA consists of a series of instantaneous descriptions (IDs), referred to as configurations. An ID reflects the entire condition of the PDA at a specific time. Although multiple formalisms are present, a standard definition of a configuration for a PDA is a triple (q,w,), where: q Q represents the current state. w $\Sigma *$ denotes the part of the input string that remains to be processed. $\Gamma *$ signifies the current contents of the stack, with the leftmost symbol of being the top of the stack. The computation starts in a starting configuration (q 0 ,w initial ,Z 0), where w initial represents the complete input string. Every stage of the calculation requires utilizing a transition rule from to shift from one configuration to another. If (q,a,A) contains (p,), and the current configuration is (q,aw,A), then the PDA can move to (p,w,). If a= , the input string w does not change. A PDA stops when it attains a configuration where no additional transitions can occur. This can happen in multiple ways: No relevant transition: The existing state, the subsequent input symbol (or), and the top stack symbol do not correspond to any specified transition rule in . Acceptance: The PDA has processed the complete input string (w=) and is in a final state (q F). This is a widely used definition of acceptance for halting. In non-deterministic PDAs, acceptance may also be determined achieving an empty stack, irrespective of the final state, or through a mixture of both methods. The selection

The acceptance condition does not change the category of languages that can be recognized non-deterministic PDAs. The idea of "maximum halting time" for a PDA, denoted (PDA), indicates the longest potential series of steps (transitions) from a starting configuration to a halting configuration for any given input string. This encompasses

-transitions, enabling the PDA to carry out internal processes that alter its state and stack without using input, which may result in exceptionally lengthy computation paths. 2.3 Decidability of the Halting Problem for Pushdown Automata A significant difference between Pushdown Automata and Turing Machines is the decidability of the Halting Problems associated with each. As noted in Section 1.2, the Halting Problem for Turing Machines cannot be decided. This indicates that no universal algorithm can ascertain if a specific TM will stop on a particular input. In sharp opposition, the Halting Problem for Pushdown Automata is solvable. This key distinction arises directly from the limited memory framework of PDAs. The Halting Problem's decidability for PDAs can be grasped examining the characteristics of their configurations and the constraints set the stack. Although a PDA's stack can increase without limit, access is confined to the top symbol (LIFO). This limitation stops a PDA from mimicking the random memory access sequences necessary for universal computation, which Turing Machines can perform. Particularly, for a specific PDA M and an input string w, the count of reachable unique configurations (q,

Chunck 5

Turing heilleika Pushdown sjálfvirkum aðgerðum og útreikningar-möguleikum pudau

Þetta skýrslan veitir nákvæma stærðfræðilega greiningu á Pushdown Automata (PDAs), með áherslu á formlega skilgreiningu þeirra, rekstrarlegar merkingar og útreikningsgetu. The halting problem's decidability for PDAs is examined, in comparison to that of Turing Machines. A key focus is the analysis of Turing completeness, showing that although single-stack PDAs lack Turing completeness, adding a second stack enhances their computational capabilities to match those of a Universal Turing Machine. The chuck subsequently establishes two new functions: (PDA), which measures the highest number of steps a specific PDA requires to stop on any input, and pudau(n), which

denotes the maximum value of (PDA) across all n-state, 2-symbol halting PDAs. presenting comprehensive arguments and utilizing the decidability of the PDA halting problem, it is shown that pudau(n) can be computed a Turing Machine, highlighting a clear difference with the uncomputability of the Busy Beaver function. This study seeks to enhance the comprehension of the theoretical boundaries and practical consequences of computation based on PDA. Pushdown Automata: Basics and Formal Structure 1.1. Pushdown Automata (PDAs) are a crucial computational model in formal language theory, mainly intended to define Context-Free Languages.

(CFLs). These languages are prevalent in computer science, constituting the foundation of programming language syntax and significantly contributing to the analysis of natural languages. PDAs offer a strong mathematical structure for generating CFLs, usually done via context-free grammars, as well as for recognizing them. The emergence of PDAs represented a notable progress in the hierarchy of computational frameworks. They successfully connect simpler Finite Automata (FAs), which can only recognize regular languages due to their limited memory, with the more powerful Turing Machines (TMs). A PDA's key characteristic is that it enhances a finite automaton with an unlimited stack memory. This additional stack offers the essential function for PDAs to manage languages featuring nested or recursive forms, a feature that is fundamentally outside the capabilities of finite automata. For example, the language of balanced parentheses, a standard illustration of a CFL, requires a stack to monitor unmatched opening parentheses, a function unachievable for a finite automaton. Aside from their theoretical importance, PDAs have wide-ranging practical uses, especially in the field of designing parsers. They play a crucial role in developing parsers for programming languages, particularly those that utilize LR(1) grammars. LR(1) grammars form a robust subclass of context-free grammars, characterized their capacity to facilitate deterministic parsing using a single lookahead symbol, which

renders them extremely expressive and broadly useful in syntactic analysis. Current studies still utilize PDAs for sophisticated computational functions. For instance, recent research like "Pre3" Dong et al. (2024) shows their continuing importance in limited decoding for large language models. This study highlights the important difference between deterministic (DPDA) and non-deterministic (NPDA) versions, especially regarding their effectiveness in real-world uses. In contemporary settings, the emphasis is frequently on enhancing runtime execution efficiency, particularly

for handling tokens that depend on context, where the fixed characteristics of DPDAs can provide notable benefits. A Pushdown Automaton M is formally described as a 7-tuple, which is a conventional mathematical framework in formal language theory that accurately represents the elements and functioning principles of the machine. This formalization is uniform throughout scholarly articles and current preprints. Sure, Elements of a PDA: The 7-Tuple (Q,Σ,Γ, ,q 0 ,Z 0 ,F) The seven elements of a PDA M are: Q: A limited collection of states. These states denote the internal setups or memory of the automaton, similar to the states in a finite state machine. The automaton moves between these states depending on its input and the contents of its stack. Σ: M . This collection includes every possible symbol that the PDA can scan from its input tape. Γ: Ɛ . This collection contains every symbol that can be held in the auxiliary stack, which acts as the unbounded memory for the PDA. : The transition function, explicitly defined as :Q×(Σ)×Γ→P(Q×Γ ∗). This function is the essence of the PDA's operational logic, governing its actions at every stage. It requires three inputs: The present state (q Q). The symbol currently being processed (a Σ). The symbol (empty string) indicates an epsilon-move, enabling the PDA to change states without requiring an input symbol. The symbol that is at the head of the stack at present (A Γ). The result of is a limited collection of potential subsequent configurations, illustrating the non-

deterministic characteristics of general PDAs. Every element in this powerset is a pair (p,), where: p Q represents the next state to which the automaton moves. Γ ∗ denotes the string of stack symbols that substitutes the removed top stack symbol A. The leftmost character of takes the place of the new top of the stack. If

= indicates that the top stack symbol A is removed without adding anything. If =A, the stack essentially stays the same. If =YZ, then Y is placed onto the stack followed Z, making Y the new top element. q 0 : The initial state, q 0 Q. This is the sole starting state where the automaton commences its computation for any specific input. Z0: The starting stack symbol, Z0 Γ. This symbol is first positioned at the very base of the stack, acting as an indicator for the start of the stack. F: The collection of accepting (final) states, F Q. This group of states is utilized for acceptance through the final state, which is one of the main methods for recognizing a language. Instantaneous Descriptions (IDs) and the Step Relation M To formalize the dynamic actions of a PDA, the notion of an Instantaneous Description (ID) is presented. An ID accurately represents the entire computational state of a PDA at any specific time. It is characterized as a 3-tuple (p,w,) Q×Σ ∗ ×Γ ∗ . p Q: Indicates the existing state of the automaton. w Σ ∗ : Signifies the segment of the input tape that remains unread. Γ ∗ : Depicts the current contents of the stack, listing the top symbol first. The transition relation formally specifies the step-relation, represented as M , on these immediate descriptions. For a specified transition rule (p,a,A,q,) , one computational step is characterized as: [(p, ax, Aγ) $\vdash_M$ $(q, x, \alpha\gamma)$]. $Thisstepholdstrueforanyleftoverinputstringxandforanyrema$ $moveifa =), andAis$

the symbol located at the peak of the stack, it may then shift to state q, read a (if a), remove A from the stack, and place the string onto the stack. Calculations of a pushdown automaton consist of sequences of these steps. These sequences are formally expressed the

reflexive and transitive closure of the step relation, symbolized as M ∗ . A computation usually starts with an initial instantaneous configuration (q 0 ,w,Z 0), where q 0 represents the starting state, w denotes the complete input string, and Z 0 is the symbol at the top of the stack. A key feature of general pushdown automata is their non-deterministic behavior. This implies that from one instantaneous description (p, w,), there could be several possible transitions as specified . The automaton can select any one of these options to proceed with its computation. This innate non-determinism greatly affects the expressive capabilities of PDAs. Deterministic Pushdown Automata (DPDAs) identify Deterministic Context-Free Languages (DCFLs), a strict subset of the Context-Free Languages (CFLs) acknowledged non-deterministic PDAs (NPDAs). This indicates that the capacity for making non-deterministic decisions enables NPDAs to identify a wider range of languages. This is a crucial difference from finite automata, as the addition of non-determinism does not increase expressive capability. The ability of an NPDA to "predict" the right computational route is crucial for identifying languages like palindromes (ww R), which are CFLs but not DCFLs. A DPDA cannot deterministically recognize the midpoint of a string to transition from pushing symbols onto the stack to popping them for comparison, a function that necessitates the "foresight" or simultaneous exploration provided non-determinism. This separation between non-deterministic capability and deterministic efficiency is a consistent theme in real-world applications such as parser design, where

The deterministic characteristics of LR(1) grammars are favored due to their lower runtime overhead, despite the loss of some theoretical expressive capability. Acceptance Criteria: Final State vs. Empty Stack Equivalence A Pushdown Automaton can recognize an input string w via two main methods. Although differing in their operational standards, these two acceptance modes are computationally equivalent, which means they identify exactly the same set of lan-

guages, specifically Context-Free Languages (CFLs). Acceptance Final State: In this mode, a PDA considers an input string w accepted if, after it has processed the entire string (meaning the remaining input w is), the automaton attains any state q that is part of its specified set of accepting states F. The state of the stack at this moment is irrelevant; the final state is what counts. Formally, the language recognized a final state, represented as L(M), is defined as: Acceptance Empty Stack: In another approach, a PDA accepts an input string w if, after it has processed the full string (w is reduced to), the stack is entirely empty (is reduced to). In this mode, the automaton's final state does not matter; acceptance is indicated solely the emptiness of the stack. Formally, the language accepted an empty stack, represented as N(M), is defined as: A key theorem in formal language theory demonstrates the equivalence of these two acceptance criteria. It asserts that for every Pushdown Automaton M that recognizes a language L(M) through a final state, an equivalent Pushdown Automaton M can be created so that L(M)=N(M), signifying that M recognizes the same language via an empty stack. In contrast, for any PDA M that recognizes a language N(M) through an empty stack, it is possible to create an equivalent PDA M such that N(M)=L(M), indicating that M recognizes the same language through a final state. This theorem is generally established in a constructive manner.

For instance, to transform a PDA that accepts final state into one that accepts empty stack, a new initial stack symbol can be introduced along with new states that, upon reaching an original final state and processing all input, continue to pop all remaining stack symbols until the newly added initial stack symbol is revealed and popped. In the same way, for the backward conversion, a new final state may be added, which is reached when the stack is empty and the input has been consumed, irrespective of the initial state. The equivalence between final state and empty stack acceptance conditions has important implications for analyzing PDA behavior, especially regarding halting. This

equivalence indicates that for every PDA, one can convert it into a corresponding PDA that stops depleting its stack. This property enables a uniform method for characterizing "halting" in PDAs. When a PDA effectively processes an input and meets one of these acceptance conditions, it is regarded as having stopped. This uniform view on halting is crucial for establishing functions such as (PDA), as it guarantees a uniform understanding of termination throughout all PDAs. From a practical perspective, this theoretical equivalence offers adaptability in PDA design and evaluation. Researchers and designers can select the most suitable acceptance mode for a specific problem without changing the established language, highlighting the resilience and flexibility of the PDA model in capturing context-free characteristics. 1.4. Deterministic vs. Nondeterministic Pushdown Automata: Expressive Power The differentiation between deterministic and nondeterministic types is a vital element of Pushdown Automata theory, greatly influencing their expressive capabilities and practical uses. A Deterministic Pushdown Automaton (DPDA) represents a limited version of a PDA in which, for any specific combination of current state q Q, input symbol a (Σ), and stack symbol A Γ, there can be at most one possible next action. An essential requirement

for determinism is that if an -move (a transition that does not use an input symbol) can occur from a configuration (q, ,A), then no input-consuming move can occur from (q,a,A) for any a Σ. This rigid requirement guarantees that the automaton's subsequent action is consistently established, removing any uncertainty or options in its calculation route. DPDAs usually accept transitioning to an accepting state upon processing the entire input. The category of languages accepted DPDAs is referred to as Deterministic Context-Free Languages (DCFLs). A well-known fact is that DCFLs constitute a strict subset of the Context-Free Languages (CFLs) acknowledged (nondeterministic) PDAs. This implies that there are CFLs that no DPDA can recognize. This strict subset relationship is a significant finding in formal lan-

guage theory, emphasizing that an NPDA's capability to "guess" the right computational path—due to having several possible transitions from one configuration—gives it more expressive power than a DPDA, which is limited to one specific choice. This "guessing" ability enables NPDAs to identify languages that necessitate unlimited lookahead or the capacity to decide only after reviewing additional input. A standard example of a CFL that is not a DCFL is the set of all palindromes, L=ww R w a,b $*$, where w R represents the reverse of the string w. A DPDA is unable to identify this language as it cannot deterministically "predict" the midpoint of the string. To check if a string is a palindrome, the automaton needs to push the initial half of the string onto the stack and then, at the midpoint, transition to popping symbols to compare with the latter half. Because the midpoint is unmarked and its position can change, a DPDA cannot deterministically decide to change modes without the risk of choosing an incorrect path. An NPDA, on the other hand, has the ability to non-deterministically

"estimate" the middle point and investigate the route that results in acceptance. DCFLs have more robust closure properties than general CFLs. For example, DCFLs are closed under both complementation and intersection with regular languages, characteristics that are not typically true for all CFLs. Additionally, every DCFL possesses a distinct context-free grammar, which indicates a singular leftmost derivation for every string. Conversely, general CFLs can be ambiguous or intrinsically non-deterministic. A context-free language recognized a deterministic PDA inherently lacks ambiguity in its parsing. The greater expressive ability of NPDAs, resulting from their non-deterministic "guessing" feature, entails a notable computational expense. Although NPDAs possess theoretical strength, their simulation frequently requires methods such as backtracking or the simultaneous exploration of various computational paths. In practical use cases, particularly for extensive data processing or in contemporary scenarios such as constrained decoding for Large Language Models (LLMs),

this may result in significant runtime delays. This is exactly why recent research initiatives, like the "Pre3" method, concentrate on creating algorithms to directly construct DPDAs from LR(1) grammars. The goal is to utilize the predictable behavior of DPDAs to discard context-sensitive tokens, thus negating the necessity for backtracking, exploratory guessing, and the upkeep of intricate persistent stacks. This significantly lowers the computational overhead for transitions, resulting in more efficient and scalable systems. This illustrates a clear trade-off: non-determinism enhances language recognition abilities, while determinism supports more efficient and feasible parsing methods. Table 1: Formal Elements of a Pushdown Automaton (PDA) Component Symbol Type Explanation States Q Finite Collection Internal storage of the automaton. Input Alphabet Σ is a finite collection of symbols read from the input tape. Stack Alphabet Γ A finite collection of symbols held in the stack. Transition Function $Q\times(\Sigma\ )\times\Gamma\rightarrow P(Q\times\Gamma*)$ determines the upcoming state and stack action according to the present.

condition, entry, and stack peak. Start State q0 q0 Q The automaton's initial state. Initial Stack Symbol Z 0 Z 0 Γ Symbol that starts at the base of the stack. Accepting States F F Q States that signify acceptance of input (for acceptance of final state). Ekspor ke Spreadsheet Tabel ini memberikan referensi yang ringkas dan langsung untuk definisi formal dari PDA, yang merupakan dasar bagi seluruh laporan. Clearly outlining each element and its mathematical classification emphasizes the thoroughness of the discourse and acts as a rapid reference for readers, guaranteeing a shared comprehension of the foundational model prior to exploring more intricate subjects such as halting and computability. Halting and Termination in Pushdown Automata 2.1. Halting for PDA Computations Defined A computation of a Pushdown Automaton essentially consists of a series of instantaneous descriptions (IDs) connected the step relation M. This sequence starts with an initial ID (q 0 ,w,Z 0) and advances through

a set of actions determined the transition function. To analyze the function (PDA), it is crucial to have a clear definition of "halting." A PDA "halts" on a particular input w if its computation sequence for w concludes after a finite number of steps. This cessation can appear in various unique forms: Acceptance: The ideal type of stopping happens when the PDA accurately identifies the input string. This can occur under two equivalent conditions: Final State: When the complete input string has been processed (i.e., the remaining input w becomes), the automaton attains a state q f that belongs to its set of accepting states F. At this stage, the content of the stack is unimportant. Formally, this relates to a configuration (q f , ,) where q f F. Empty Stack:

Upon processing the full input string (w turns into), the automaton's stack is entirely cleared (is now). In this scenario, the particular state of the automaton does not serve as a criterion for acceptance. In formal terms, this relates to a setup (q, ,). As noted in Section 1.3, these two modes of acceptance are equivalent regarding the class of languages they define. Rejection/Dead End: A computation stops when the PDA reaches a state with no possible valid transitions left, and it has not met any acceptance criteria. This happens when there is no defined transition in for the current state, input symbol (or), and the symbol on top of the stack. In this situation, the computation ends, yet the input string is not accepted. Non-Halting (Infinite Loop): A PDA fails to halt on an input if its processing enters into an infinite loop. This indicates that the automaton persistently moves between configurations without fully processing the input, depleting its stack, or hitting an accepting state. This calculation continues endlessly, and the input is implicitly dismissed. In the specific definition of (PDA)=maxt N w $\Sigma *$ such that M halts on input w in exactly t steps, "halts on input w" pertains solely to computations that finish, either through acceptance or arriving at a dead end, as long as they do not engage in an infinite loop. The codomain N for (PDA) is essential,

as it indicates that the quantity of steps t has to be a finite natural number. This indicates that only computations that finish add to the set from which the maximum is obtained. If a PDA is able to operate without limit on a specific input, the computation time for that input is excluded from (PDA). Additionally, if the collection of finite halting times for a specific PDA is unbounded (for instance, if it accepts an infinite language in which computation duration

increases with input length), then (PDA) would be unbounded, contradicting its definition as a natural number. This suggests that (PDA) is only properly defined (i.e., finite) for PDAs where the collection of maximum halting steps is actually limited, which generally aligns with PDAs that recognize finite languages. Sure, Decidability of the Halting Problem for Pushdown Automata The halting problem, a fundamental concept in computability theory, inquires whether a universal algorithm can be created to ascertain, for any specific program (or automaton) and any input, if that program will ultimately halt or operate indefinitely. Theorem: The halting problem for Pushdown Automata is solvable. This is a key characteristic that distinctly separates PDAs from stronger computational models, particularly Turing Machines. Contrast with Turing Machines: A Key Difference The halting problem for Turing Machines is well-known to be undecidable, a significant finding made Alan Turing. This undecidability indicates that it is impossible to create a universal algorithm that can accurately ascertain, for any given Turing machine and input, whether that machine will eventually stop. The fundamental reason for this undecidability is the Turing Machine's limitless and unregulated memory access (its infinite tape), enabling complex, recursive, and unpredictable computation paths. A Turing machine is capable of simulating any algorithm, and its tape can hold an infinite amount of information, which can be retrieved and altered at any time. This adaptability facilitates the creation of self-referential programs that result in the undecidability proof via contradiction. In sharp contrast, a single-stack PDA, al-

though having unlimited memory via its stack, is essentially limited its Last-In, First-Out (LIFO) approach to memory access. This limitation is crucial for the decidability of its halting issue. Although a PDA's stack can expand indefinitely, its operations (push and pop) are restricted to the top.

element. This restricted access pattern stops the type of random memory alteration that defines Turing Machines and results in undecidability. For models that possess limited memory, like finite automata, the halting problem is clearly decidable due to having a finite count of possible states; any computation must either stop or revisit a configuration, which is identifiable. Even though PDAs possess unlimited memory, the distinct LIFO access pattern they utilize enables a comparable, though more intricate, argument regarding decidability. The hierarchy of computational power further emphasizes this idea. The halting problem for a one-state Turing machine is decidable, mainly due to its memory being practically finite without the capability to alter states. A Turing machine with only two states can achieve universality, making its halting problem undecidable. This emphasizes that even slight enhancements in the power of a Turing machine model rapidly result in undecidability, highlighting the distinctive role of PDAs as a framework with limitless memory yet a decidable halting problem. The LIFO restriction on memory access is the key element that hinders PDAs from attaining the complete computational power of Turing Machines and, therefore, maintains their halting problem in the domain of decidability. Constructive Proof of PDA Halting Decidability The PDA halting problem's decidability can be illustrated proving that every infinite computation will ultimately revisit a "relevant" configuration in a manner that enables identification. The computation of a PDA can be seen as a route through a graph of instantaneous descriptions (IDs). A computation path that is infinite suggests that there is a cycle present in this graph. The difficulty lies in the fact that the content of the stack can be of any size,

resulting in an infinite total number of IDs. Nevertheless, the LIFO characteristic of the stack restricts the methods through which infinite loops might happen without ultimately resulting in a stop. An endless

In a PDA, computation must entail a series of -moves (transitions that do not utilize input) that either maintain the stack height or raise it without limit. Should the stack height reduce endlessly, it would ultimately be vacant, resulting in either a stoppage (acceptance with an empty stack) or a deadlock (if no transitions exist for an empty stack). A formal argument for decidability is based on noting the finite nature of the PDA's internal state and the top symbol on the stack. If a PDA becomes trapped in an infinite loop, it will inevitably encounter a pair (q,A) again, where q represents the current state and A denotes the symbol at the stack's top, with the rest of the input string remaining the same. If a pair (q,A) is visited again and the net result of the transitions between the two instances of (q,A) results in a stack height that is non-decreasing (meaning the stack does not decrease from when the pair was first seen, or it increases), then the PDA is caught in an infinite loop. This is due to the fact that the section of the stack beneath A remains unchanged the actions on A and above, and the input tape is not progressing. If the stack height remains non-decreasing during this cycle, the automaton can endlessly repeat the cycle without consuming input or depleting the stack. Algorithms can identify these cycles. One method includes monitoring the configurations that have been visited, particularly concentrating on the (q,A) pairs along with the stack's length. Specifically, for any specific input string w, the count of unique configurations (q,suffix of w,stack content) is limitless. Nonetheless, the key point is that when a PDA is in state q with A at the top of the stack, and it executes a series of -moves that brings it back to state q with A remaining on the

if the stack remains at its maximum height and has not reduced, then this sequence can continue indefinitely. Given that there is a

limited quantity of states (Q) and stack symbols (Γ), there exists a finite number of pairs (q,A). If the PDA executes a long enough sequence of -moves without processing input or changing the stack top in a way that disrupts cycles, it will ultimately have to repeat a (q,A) pair. If this repetition happens with a stack that is at least as long as before, an infinite loop is identified. This enables the development of a decision procedure: execute the PDA incrementally, and if a configuration recurs under situations that ensure an infinite loop (for example, identical state, same top-of-stack, and the segment of the stack below remains unchanged in a manner that would necessitate termination), it can be concluded that the PDA will not halt. Otherwise, it will ultimately come to a stop. This decidability arises directly from the LIFO memory limitation, which obstructs the intricate, non-local memory access patterns that cause Turing Machines' undecidability. The Turing completeness of pushdown automata is a fundamental idea in computability theory, determining the highest level of computational ability that a machine can reach. A computational model is deemed Turing complete if it has the capability to replicate any Turing Machine, and thus, it can calculate any computable function or identify any recursively enumerable language. This section explores the question: "are pushdown automata Turing complete?". A computational model M is considered Turing complete if it is capable of simulating any given Turing Machine (TM). This indicates that for every TM T and every input w, there is a program or setup for M that operates exactly like T on w, encompassing both halting behavior and output. Similarly, a Turing complete model can identify all recursively enumerable

if the stack remains at its maximum height and has not reduced, then this sequence can continue indefinitely. Given that there is a limited quantity of states (Q) and stack symbols (Γ), there exists a finite number of pairs (q,A). If the PDA executes a long enough sequence of -moves without processing input or changing the stack

top in a way that disrupts cycles, it will ultimately have to repeat a (q,A) pair. If this repetition happens with a stack that is at least as long as before, an infinite loop is identified. This enables the development of a decision procedure: execute the PDA incrementally, and if a configuration recurs under situations that ensure an infinite loop (for example, identical state, same top-of-stack, and the segment of the stack below remains unchanged in a manner that would necessitate termination), it can be concluded that the PDA will not halt. Otherwise, it will ultimately come to a stop. This decidability arises directly from the LIFO memory limitation, which obstructs the intricate, non-local memory access patterns that cause Turing Machines' undecidability. The Turing completeness of pushdown automata is a fundamental idea in computability theory, determining the highest level of computational ability that a machine can reach. A computational model is deemed Turing complete if it has the capability to replicate any Turing Machine, and thus, it can calculate any computable function or identify any recursively enumerable language. This section explores the question: "are pushdown automata Turing complete?". Sure, A computational model M is considered Turing complete if it is capable of simulating any given Turing Machine (TM). This indicates that for every TM T and every input w, there is a program or setup for M that operates exactly like T on w, encompassing both halting behavior and output. Similarly, a Turing complete model can identify all recursively enumerable languages and calculate all partial recursive functions. The Church-Turing thesis asserts that any function that can be computed via an algorithm is also computable a Turing Machine, establishing Turing completeness as the theoretical limit for effective computability. 3.2. Computational Capability of Single-Stack Pushdown Automata A key finding in formal language theory indicates that a single-stack Pushdown Automaton lacks Turing completeness. PDAs have strictly lesser power compared to Turing Machines. Although PDAs enhance finite automata adding an unlimited stack,

their memory usage adheres to a Last-In, First-Out (LIFO) principle. This LIFO restriction restricts the kinds of calculations a PDA can execute. The set of Context-Free Languages (CFLs) is exactly the class of languages that (non-deterministic) Pushdown Automata can recognize. The Chomsky hierarchy ranks CFLs higher than regular languages, yet distinctly lower than recursively enumerable languages (those recognized Turing Machines) and context-sensitive languages. This ordered hierarchy suggests that some languages are recursively enumerable yet not context-free, meaning they cannot be recognized a single-stack PDA. An example of a language that is recursively enumerable but not context-free is L=a n b n c n n 1. For this language to be identified, a machine must confirm that the number of 'a's, 'b's, and 'c's are all the same. One stack can serve to measure the quantity of 'a's against 'b's (for instance, pushing 'a's and popping for 'b's). Nonetheless, after assessing 'a's and 'b's, the stack would either be empty or hold a certain marker, resulting in the loss of the data needed to compare with 'c's. Conversely, if 'a's are advanced and 'b's are paired, then 'c's need to be paired with the leftover stack elements. However, the LIFO characteristic indicates that the 'a's are positioned at the bottom. To align 'c's with 'a's, it is necessary to retrieve the 'a's that were initially added, which is impossible without compromising the 'b' data or the stack configuration. This demands a type of memory retrieval that exceeds what a single LIFO stack can handle. Languages such as a n b n c n require the capability to compare three amounts, something that a single stack cannot handle at once or in succession without losing crucial information. The constraint arises from the PDA's lack of capacity to "peek" or retrieve specific elements located deep in the stack without removing the elements that are above them. This stands in stark contrast to a Turing Machine's tape, which permits random access to any cell, allowing it to store and retrieve information much more flexibly. Consequently, a single-stack PDA lacks Turing completeness; it is unable to simulate any arbitrary

Turing Machine due to its overly restrictive memory model. 3.3. The Turing Completeness of Multi-Stack Pushdown Automata Although a single-stack PDA lacks Turing completeness, its computational capability significantly enhances with the incorporation of an additional stack. A Pushdown Automaton with two stacks has the same computational power as a Turing Machine and is therefore Turing complete. This is a noteworthy outcome that shows how a seemingly minor enhancement in memory architecture can result in a substantial advancement in processing capability. The equivalence comes from the ability of two stacks, functioning with LIFO principles, to together replicate the actions of one unbounded Turing Machine tape. A Turing Machine tape can be imagined as a linear array of cells, with a read/write head located at a single cell. The head can shift to the left or right and either read or write symbols where it is currently located. To emulate a Turing Machine T=(Q T ,Σ T ,Γ T , T ,q 0T ,B,F T) with a two-stack PDA M=(Q M ,Σ M ,Γ M , M ,q 0M ,Z 0 ,F M), the two

Stacks are utilized as such: Stack 1 (Left Stack): Holds the elements of the Turing Machine tape to the left of the head, with the character directly to the left of the head at the top. Stack 2 (Right Stack): Holds the contents of the Turing Machine tape located to the right of the head, with the symbol directly next to the head positioned at the top. The present cell beneath the TM head is theoretically maintained within the PDA's finite control. Let's elaborate on the simulation of one TM move: Starting Setup: The input string w for the Turing Machine is first positioned on the right stack (Stack 2), alongside a unique bottom-of-stack marker. The left stack (Stack 1) starts out devoid of elements, apart from its own bottom-of-stack indicator. The PDA begins in its starting state, which aligns with the TM's starting state, and with the TM head conceptually positioned at the leftmost character of w. Imitating a TM action (read, write, shift): Assume the Turing Machine T is in state q T , reads symbol X from the tape, writes Y, and shifts Left (L) or Right (R). Case 1: TM shifts Right (R). The

PDA removes the top symbol from Stack 2 (which indicates the symbol presently beneath the TM head). Let this be X. The PDA's finite control alters its state according to q T and X. The PDA places Y (the symbol written TM) onto Stack 1 (the left stack). The PDA transitions its state to the new TM state. The updated symbol beneath the TM head is now the previous peak of Stack 2. This symbol successfully transfers from Stack 2 to the PDA's finite control for the subsequent step. This action successfully shifts the "head" one space to the right on the emulated tape. Caso 2: TM

shifts Left (L). The PDA removes the top symbol from Stack 1 (which indicates the symbol directly to the left of the TM head). Allow this to be X . The finite control of the PDA changes its state according to q T and X . The PDA places Y (the symbol inscribed TM) onto Stack 2 (the stack on the right). The PDA transitions its state to the new TM state. The updated symbol beneath the TM head is now the former peak of Stack 1. This symbol successfully transitions from Stack 1 to the PDA's finite control for the subsequent step. This action successfully shifts the "head" one spot to the left on the emulated tape. Managing Blanks: When a stack empties during a transition (for instance, moving left from the leftmost non-blank character), the PDA can theoretically "interpret" a blank symbol B and add B to the other stack as required, mimicking the infinite blank tape. Halting: When the Turing Machine stops (enters an accepting state), the two-stack PDA similarly enters an accepting state. This simulation shows that every calculation executed a Turing Machine can be duplicated a two-stack PDA. A two-stack PDA can emulate any Turing Machine, granting it equal computational capability and making it Turing complete. This outcome emphasizes that the main drawback of a single-stack PDA is its lack of access to random memory locations, a challenge that is addressed utilizing two stacks that can efficiently "exchange" information, resembling the two-way movement and random access abilities of a Turing Machine's tape. Examination

of (PDA) and pudau(n) This part explores the mathematical characteristics of the two functions outlined in the user inquiry, (PDA) and pudau(n), and discusses the Turing Machine's ability to compute pudau(n). Formal Definition and Explanation of (PDA)

The function is described as: $[\sigma : x \mid x$ is a pushdown automaton $\rightarrow$ $\mathbb{N}]\ Thisdefinitionindicatesthatforaspecific PushdownAutomaton M, (M)representstth$ $negativeinteger.Thislimitationonthecodomainisessentialandimposesanunspokenlimit$ $anbnn1), thestepsneededtoacceptastringgenerallyincreasealongsidethestring's length.F$ $emptyforanyPDAwithintheeffectivedomainof.Thisrequirementismetexactly PDAst$

and stop on at least one input. 4.2. Formal Definition and Interpretation of pudau(n) The function pudau is expressed as: [pudau : N$\rightarrow$ $\mathbb{N}][n \mapsto$ pudau$(n)]$ $Thefunctionpudau(n)acceptsanaturalnumbern(denotingthecountofs$ $state, 2-symbolPushdownAutomatathat"stop".Theterm"PDAthathalts"inthiscontext$ $state, 2-symbolPDAswheretheseofhaltingcomputationlengthsislimited, i.e., PDAsth$ $EnumerationofPDAs : Foranyintegern, thequantityoffeasiblen-$ $state, 2-symbolPushdownAutomataislimited.APDAischaracterizedits7-$ $tuple(Q, , , , q, Z, F).ThequantityofstatesQ = nremainsconstant.Theinputalphabet =$ $2remainsconstant.Thestackalphabetcanbeselectedtohaveaspecificsize(forinstance, 2+$ $1fortheinputsymbolsandabottom-of-stackmarker, orasmuchasn2dependingontransit$ $QZ0orasizethatislimitedbasedonderivation).$

from n). The transition function :Q$\times(\Sigma\ )\times\Gamma\rightarrow$P(Q$\times\Gamma*$) possesses a finite domain (count of tuples (q,a,A)) and a finite range for each tuple (finite subsets of Q$\times\Gamma*$, where $\Gamma*$ may be limited to strings of restricted length, commonly 0, 1, or 2 symbols for practical PDAs, or a length corresponding to n). Considering these finite elements, the overall count of unique n-state, 2-symbol PDAs is limited. A Turing Machine can methodically list and create a description for every one of these finite PDAs. Determining if a PDA Recognizes a Finite Language: For each listed PDA M, the following step is to assess whether it recognizes a finite language. This is a solvable issue in formal language theory. There is an algorithm that can identify if a specific PDA accepts either a finite or infinite language. This generally entails

examining the reachability graph of the PDA's states and identifying cycles that can be navigated an unlimited number of times while processing input or expanding the stack indefinitely. If these "pumping" cycles are present, the language is infinite; if not, it is finite. A Turing Machine is capable of carrying out this decision process for every listed PDA. Calculating (PDA) for PDAs with Finite Languages: If a PDA M is found to recognize a finite language, then the collection of strings it accepts is also finite. As a result, there exists a longest string within the set L(M). For every such string, the steps taken M to stop are limited. Given that the language is finite, the highest count of steps in all halting computations for M (i.e., (M)) must likewise be finite and clearly defined (according to the explanation in Section 4.1). A Turing Machine can calculate (M) : Producing all potential input strings w Σ $*$ up to a defined maximum length (which is limited for finite languages). For every

produced w, execute the PDA M on input w. Because the halting problem for PDAs is decidable (as mentioned in Section 2.2), the simulation will either stop (accept or reject) or be identified as an infinite loop. If it stops, note the number of steps t. Upon simulating all pertinent inputs, identify the highest t observed. This maximum t equals (M). Calculating pudau(n): After (PDA) is determined for every n-state, 2-symbol PDA that recognizes finite languages (and thus "halts" according to the definition), the Turing Machine can straightforwardly select the highest value from all the computed (PDA) values. Given that there is a limited quantity of these PDAs for a specific n, and each (PDA) value is a finite natural number, determining the maximum of this limited set of natural numbers is a simple computable task. Thus, a Turing Machine can compute pudau(n). The computability of pudau(n) sharply contrasts with the notorious uncomputability of the Busy Beaver function, BB(n). This comparison emphasizes a basic distinction in the core computational models. The Busy Beaver function, BB(n), represents the highest number of steps executed any n-state,

2-symbol halting Turing Machine that begins on an empty tape. Although their definitions appear similar (both focus on maximizing steps for n-state machines), BB(n) is well-known to be uncomputable. This uncomputability stems directly from the undecidability of the halting problem concerning Turing Machines. To calculate BB(n), one would, in theory, have to: List all n-state, 2-symbol Turing Machines. (This is a limited quantity, akin to PDAs). For every machine, assess whether it stops. This is exactly the halting problem that cannot be decided. Given that no universal algorithm exists to ascertain whether a specific Turing Machine will halt, there is no means to

remove the non-halting machines from the collection. Even if it were possible to identify all halting machines, there is no computable limit on the number of steps a halting Turing Machine could execute. If such a limit were to exist, it would indicate a resolution to the halting problem (execute the machine for that number of steps; if it hasn't stopped, it won't ever stop), which creates a contradiction. Essentially, the uncomputability of BB(n) stems from the unrestricted access to memory and the universal computational capabilities of Turing Machines, resulting in the undecidability of their halting characteristics. On the other hand, the computability of pudau(n) directly results from the decidability of the halting problem concerning Pushdown Automata. The LIFO memory restriction of PDAs, which confines their computational capability to Context-Free Languages, is exactly what renders their halting behavior both predictable and decidable. Since determining if a PDA halts for a specific input is decidable, and importantly, whether a PDA recognizes a finite language (which indicates a limited number of steps for any halting computation), a Turing Machine can methodically evaluate all pertinent PDAs and calculate their maximum halting durations. This essential distinction in the decidability of the halting problem fundamentally explains the significant disparity in computability between pudau(n) and the Busy Beaver function. Conclusions This document has offered an extensive mathemati-

cal examination of Pushdown Automata, their key characteristics, and the computability of the new functions (PDA) and pudau(n). Pushdown Automata are rigorously defined a 7-tuple that includes states, input and stack alphabets, a transition function, a starting state, an initial stack symbol, and accepting states. Their operational semantics are accurately represented instantaneous descriptions and a relation of steps. A fundamental trait is their non-deterministic quality, which enables them to identify all Context-Free Languages, a category that is strictly

more potent than Deterministic Context-Free Languages identified DPDAs. This variance in expressive capability stems from the non-deterministic PDA's ability to "predict" computational paths, a feature that, although theoretically strong, may result in heightened computational complexity in real-world parsing situations. The equivalence between acceptance through final state and acceptance via empty stack streamlines the analysis of PDA behavior, offering a cohesive viewpoint on their termination. Concerning the questions asked: Is Pushdown Automata Turing complete? A single-stack Pushdown Automaton lacks Turing completeness. Its computational capabilities are confined to identifying Context-Free Languages, which constitute a proper subset of the recursively enumerable languages acknowledged Turing Machines. This restriction arises from the LIFO (Last-In, First-Out) memory access limitation of its sole stack, which hinders the random memory manipulation necessary for universal computation. Nevertheless, the computational capabilities of Pushdown Automata significantly enhance with extra memory. A Turing complete system can be achieved with a Pushdown Automaton that has two stacks. These two stacks can efficiently replicate the bidirectional, random-access tape of a Turing Machine, thus obtaining the same computational capabilities. Is pudau(n) computable using a Turing machine? The Turing Machine can compute the function pudau(n). The definition of (PDA) as a mapping to N indicates that it is only properly defined for PDAs with

a bounded set of halting computation lengths, corresponding to PDAs that recognize finite languages. The computability of pudau(n) arises directly from two essential characteristics: The collection of n-state, 2-symbol Pushdown Automata is finite and can be reliably listed a Turing Machine. The halting problem for Pushdown Automata is solvable. This indicates that for any specific PDA and input, a Turing Machine can ascertain if the PDA will stop or operate endlessly. Moreover, it is possible to determine if a specific PDA recognizes a finite language. A Turing Machine can calculate pudau(n) going through

every conceivable n-state, 2-symbol PDA. For every such PDA, it can ascertain whether it identifies a finite language. Should it do so, the Turing Machine is able to calculate its (PDA) value simulating it across all pertinent inputs (which have limited length due to the finite language property) and logging the highest number of steps taken for any halting computation. Ultimately, the highest of these finite (PDA) values is chosen to produce pudau(n). This computability is in sharp contrast to the uncomputability of the Busy Beaver function, BB(n), which assesses the maximum steps for Turing Machines with n states. The key distinction is in the decidability of the halting problem for PDAs compared to its undecidability for Turing Machines. This differentiation highlights the essential importance of memory access patterns and the constraints of computational models in defining the limits of computability.

Chunck 6

Niðurstöðu

A Turing Machine (TM) M getur í raun útreiknað stoppvandamálið fyrir ekki-Turing fullkomin sjálfvirk. For a Finite Automaton (FA) $A = (Q, \Sigma, \delta, q_0, F)$, every input $w \in \Sigma^*$ results in a finite computation, meaning A always terminates. A TM easily resolves this. For a Pushdown Automaton (PDA) $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, the issue of halting is decidable. A TM can replicate P's states (q, w, γ) and identify recurrent configurations (q,

Chunck 7

Endalok

Let Pudau – Puhoy =

pudau(23028)

I have found peace in the notion of void– who knows that none shall ever truly break free From the dream of the world and exist without illusion. In dreams I have learned from Lethe's brink– Aware the driven dust will flow again When the grave has shattered its locked and stone oath. And I

have grinned at the inactivity of affection– Who discover the blades of your remembered charm placed in front, behind, and over me. Delightful paradox is the essence of each woman. As I burned with love, expressed fervent words, heart complete, She chuckled, but when I glanced away To catch another's yearning gaze, From which emerged an unspoken realm, She immediately positioned herself between us, grasped My hand in hers, her chest heaved softly, Her eyes were shyly lowered for a moment, Then her lips were raised towards my face. I WITNESSED a man ready to compose a poem: He stomped harshly through a subway car, Behind him, both sides. Smashed corn and tormented expressions. Twenty minutes later, he composed a verse of re-

markable delicacy. In Anster, ages ago, in the Fife shire, There dwelled a man seeking a wife to admire; A bold fisher was he, rugged and brave, With a temper much hotter than cold waves; And he frequently declared that whoever was wed Should resemble his trim wherry instead, Compliant in all matters to the rightful might With which he, as steersman, would guide her in flight, And that with her words she ought not to vex him, Nor for her life dare once to dissent from him, But whatever he did or proclaimed, She should view as law, and let it be framed. Eventually, as events unfolded, this fisherman learned of a girl from Dysart, Who was very humble, gentle, and shy, With lips that seemed hardly able to reply; In essence, just what he desired; Thus, love was quickly proposed and quickly accepted. The couple got married, and he proudly sailed home with his sweet bride; yet, while they were navigating the coastline, he observed what he had never noticed; For she, when prompted the merry bridegroom To shift aside slightly for space, Had hardly shifted from her spot a moment, When he perceived her back in it; And this in such a calm and gentle manner, As if she both would and would not comply. The deed was minor—could cause no distress, Yet still it stirred within him some unease. And quickly its significance became clear, And soon our fisherman started to fear, For before the first month had passed away, The lady's voice altered in its sway, And to all that he spoke or did She opposed him outright; Till realizing how thoroughly he was hindered, He vowed he was nearly broken-hearted. A tumultuous year had slipped , When one morning at dawn's early light, To set out fishing, he discovered his boat, Which just the previous night he had left afloat, Was gone and lost, vanished into the sea, Leaving merely a stump of the rope on the quay. An unfriendly neighbor was responsible— This was at least a belief among our fishermen. He rushed back home to share with his wife How a scoundrel had freed his boat with a knife; The rope stump he held up as he declared— "My dear," she replied, "you mean with shears; The cut is distinctly clear and evident," "A knife," said he, the husband

insisted. And to it they proceeded with this pleasant affair, Until he vowed she'd end his life with her chatter; And finally infuriated beyond all bearing, He declared he would test her with a fresh way of preparing; So directly to the dock he pulled her along, And, grasping her the edges of her dress, He plunged her into the overflowing stream, And held her there for quite a while; Then pulling her up, he inquired how She believed the rope had been cut now? She exclaimed, breathless, " shears, shears!"— And once more she plunged headfirst. This time he promised she would taste the salty water before he pulled her up; thus, he held her down for at least two minutes, so that when she resurfaced, all words had vanished. Yet, yet, nonetheless, whether vocal or silent, She would not give in to her partner: Thus, when the inquiry was repeated About her views on how the cord was severed. She merely raised her fingertip, And pointed it at the next play clip— A gesture that clearly, no matter how he might sway, She shared the same view today. That signal marked the cut of Destiny's shears, For once more she was immersed her partner, And allowed to drift on the estuary, And was never again witnessed on land. Now all you wives from Stirling to Crail, Heed this mournful story in detail, And if with your husbands you want to sway, Ensure that you don't the plan betray; Avoid doing things that they disdain, Or say words that will urge them to explain; You can lead, I promise, without constriction, But never, oh! never, through contradiction.

.